Bericht des Instituts für Aerodynamik und Strömungstechnik

Report of the Institute of Aerodynamics and Flow Technology

IB 124-2014/910

Eine TAU-Simulink-Kopplung zur aerodynamisch-flugdynamischen Simulation – Implementierung und Anwendung auf eine generische militärische Transportflugzeugkonfiguration mit Beschädigung

Lars Reimer, Christian Gall, Sven Geisbauer

Institut für Aerodynamik und Strömungstechnik (AS)
Institut für Flugsystemtechnik (FT)
Braunschweig

Herausgeber:
Deutsches Zentrum für Luft- und Raumfahrt e.V.
in der Helmholtz Gemeinschaft
Institut für Aerodynamik und Strömungstechnik
Lilienthalplatz 7, 38108 Braunschweig

Stufe der Zugänglichkeit: 2
Braunschweig, im Juli 2014

Institutsdirektor:
Prof. Dr.-Ing. habil. C.-C. Rossow

Verfasser:
Dipl.-Ing. Lars Reimer (AS)
Dipl.-Ing. Christian Gall (FT)
Dipl.-Ing. Sven Geisbauer (AS)

Abteilung: $C^2A^2S^2E$
Abteilungsleiter:
Prof. Dr.-Ing. N. Kroll

Der Bericht enthält:
89 Seiten
31 Bilder
8 Literaturstellen

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund

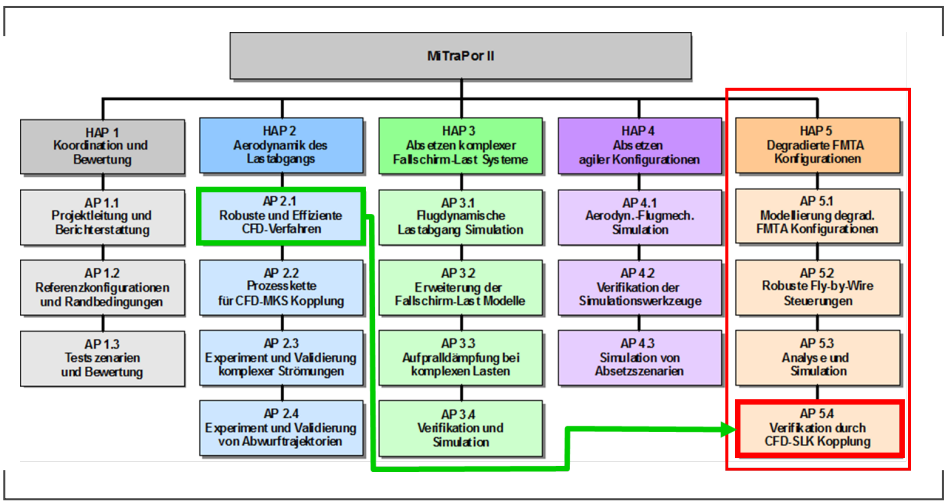
Die im vorliegenden Bericht dargestellten Arbeiten entstanden im Rahmen des DLR-Projekts MiTraPor II [1] im Arbeitspaket (AP) 5.4 (siehe Organigramm von MiTraPor II in Abb. 1.1). MiTraPor II lief im Zeitraum 2010-2013. Es folgte auf das Projekt MiTraPor (2006-2009).

In MiTraPor II wurden primär Simulations- und Bewertungswerkzeuge entwickelt, mit denen neue Technologien im Zusammenhang mit militärischen Transportflugzeugen virtuell erprobt und optimiert werden können. Neben der Simulation von Lastabsetzvorgängen über die Heckrampe eines Transportflugzeugs lag ein anderer Schwerpunkt des Projekts in der Simulation von Konfigurationen, die nachfolgend als degradierte Konfigurationen bezeichnet werden und deren Flugleistungen oder -eigenschaften infolge eines Fehlers oder einer Beschädigung eingeschränkt bzw. ungünstig verändert sind. Dies dient folgendem Zweck. Bei Fehlerfällen oder Beschädigungen kann ein militärisches Transportflugzeug aufgrund einer Bedrohungslage nicht immer sofort landen oder zum Startflughafen zurückkehren bzw. muss in einem eingeschränkten Flugkorridor bleiben. Die Erfüllung der militärischen Missionsziele besitzt oftmals einen so hohen Stellenwert, dass eine Fortsetzung der Mission trotz Beschädigung notwendig ist, sofern die Schwere der Beschädigung dies zulässt. Vor diesem Hintergrund ist die Beantwortung der Frage, ob eine degradierte Konfiguration sicher fliegen, starten und landen kann bzw. bis zu welchem Beschädigungsgrad dies möglich ist, von hoher Bedeutung.

Die Analyse typischer Fehlerfälle ist im realen Flugversuch allerdings nur sehr schwer oder aufgrund des Sicherheitsrisikos überhaupt nicht möglich. Daher ist die numerische Modellbildung kritischer Fehlerfälle (insb. des Ausfalls von Steuerflächen und/oder der strukturellen Beschädigung an Höhen- und Seitenleitwerk), sowie die Simulation der Einflüsse von Beschädigungen auf Flugleistungen und -eigenschaften besonders wichtig.

Um die Auswirkungen von einsatzrelevanten Beschädigungen auf das Flugverhalten analy-

Abbildung 1.1: **Organigramm des Projekts MiTraPor II. Die Arbeiten des vorliegenden Berichts sind im rot markierten Arbeitspaket 5.4 entstanden.**



sieren zu können bzw. geeignete Fehlerkompensationsmechanismen zu entwickeln, wurden in MiTraPor II dazu geeignete numerische Werkzeuge entwickelt. Das in diesem Bericht vorgestellte Verfahren wurde vor diesem Hintergrund entwickelt. Das Verfahren besteht aus einer Kopplung zwischen einem hochgenauen aerodynamischen Simulationsverfahren — dem DLR TAU-Code [2, 3]) — , das maßgeblich vom *DLR-Institut für Aerodynamik und Strömungstechnik* entwickelt wird, und einem MATLAB/Simulink-Flugdynamikmodell inkl. Aktuatorik und Regelungssystemen eines generischen militärischen Transportflugzeugs — dem sogenannten FMTA (siehe nachfolgenden Abschnitt 1.2). Das Flugdynamikmodell des FMTA wurde vom *DLR-Institut für Flugsystemtechnik* u. a. im Rahmen des Vorgängerprojekts MiTraPor entwickelt und in MiTraPor II weiterentwickelt.

1.2 Das erweiterte FMTA-Modell

Im Vorgängerprojekt MiTraPor wurde ein Modell eines repräsentativen modernen Militärtransportflugzeugs entwickelt, mit dem die im Projekt erstellten Bewertungsverfahren evaluiert und getestet werden konnten. Dieses entstandende Modell wurde als *Future Military Transport Aircraft* (FMTA) bezeichnet.

Es handelt sich um ein echtzeitfähiges flugdynamisches Modell, das in der graphisch orientierte Programmiersprache MATLAB/Simulink implementiert wurde. Die Aerodynamik

dieses FMTA-Modells basiert auf Derivativen, die in Windkanalversuchen ermittelt wurden. Für die Modellierung der Einflüsse von Propellerströmung und Hochauftriebssystem wurden Derivative aus CFD-Berechnungen genutzt.

Der FMTA-Modellblock wurde in eine Blockbibliothek integriert und kann je nach Aufgabenstellung in verschiedenen Arbeitsumgebungen eingesetzt werden. Mit einem zentral gespeicherten Flugzeugmodell lassen sich so verschiedene Simulationsaufgaben wie z. B. Trimmrechnung oder flugdynamische Simulation durchführen. Mit der Trimmrechnung lassen sich die notwendigen Steuergrößen und Anfangswerte der Zustandsgrößen für stationäre Flugzustände wie z. B. den unbeschleunigten horizontalen Geradeausflug bestimmen. Für weitere Details zum FMTA-Modell wird auf den Abschlussbericht von MiTraPor II verwiesen [1].

Im Rahmen dieses Berichts wurde das zum Ende von MiTraPor II vorliegende erweiterte FMTA-Modell genutzt. Die dem FMTA zu Grunde liegende Geometrie ist in Abb. 1.2 in einer Aufsicht, Seitenansicht und aus einer perspektivischen Ansicht dargestellt. Die Lagen und Größen der in Abb. 1.2 gezeigten Steuerflächen (rot markierte Oberflächenanteile) wurden ebenfalls generisch gewählt. Sie sind lediglich an die Eigenschaften der Steuerflächen bei bestehenden Militärtransportern vergleichbarer Bauart angelehnt.

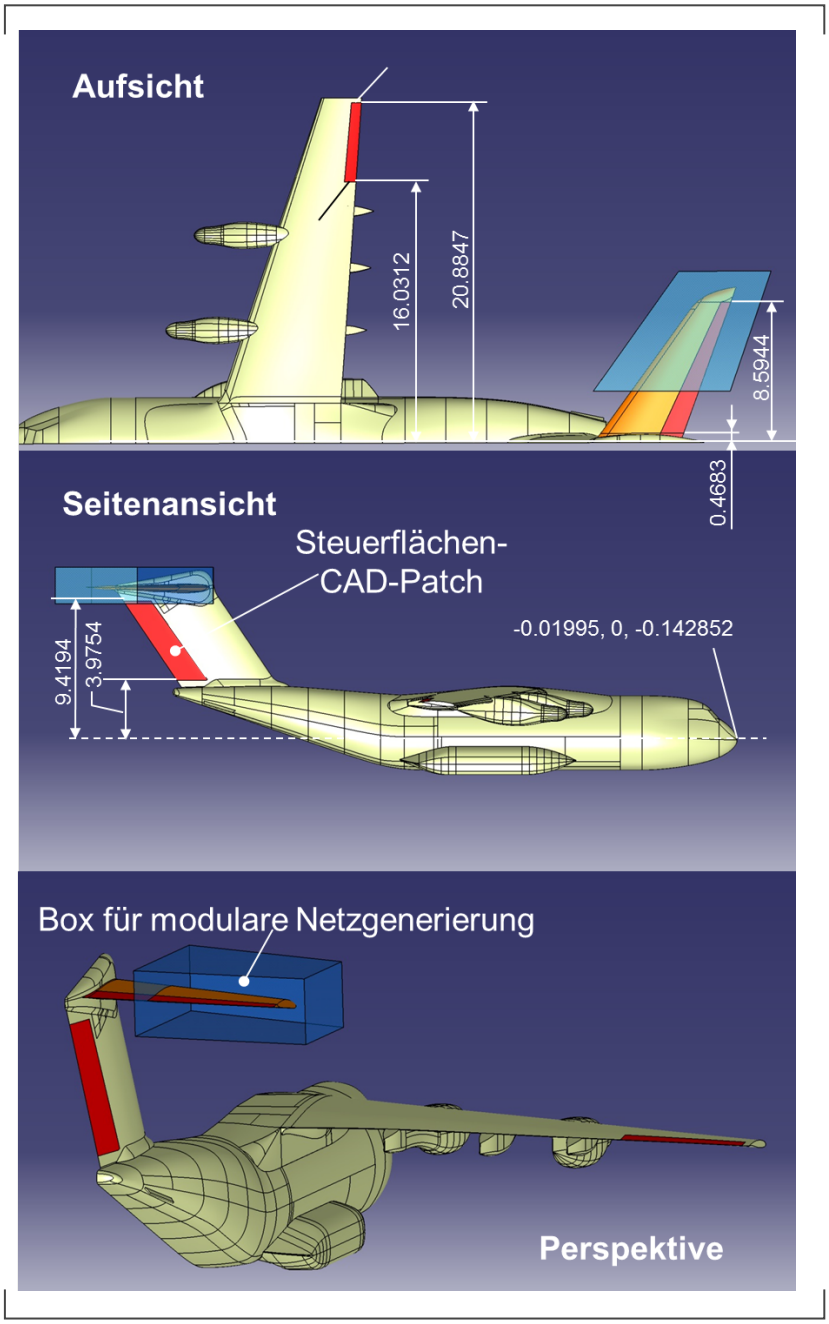
1.3 Zielsetzung

CFD-Simulationen können die reale Aerodynamik des FMTA — insbesondere im Falle einer strukturellen Beschädigung — wesentlich präziser abbilden, als dies das auf Derivativen basierende Aerodynamikmodell kann, welches im MATLAB/Simulink-Modell des FMTA bis dato verwendet wurde.

Dementsprechend bestand in MiTraPor II u. a. die Zielsetzung, das vereinfachte Aerodynamikmodell im Simulink-Modell des FMTA durch Kopplung mit dem hochgenauen CFD-Simulationscode TAU zu ersetzen. Dazu musste eine Schnittstelle geschaffen werden, über die (a) die CFD-Simulation von der in Simulink stattfindenden Flugdynamiksimulation in jedem Kopplungsschritt den aktuellen Flugzustand des FMTA, sowie dessen Steuerflächenausschläge und Triebwerkseinstellungen erhält, und (b) die CFD-Simulation die jeweils aktuell auf das FMTA einwirkenden Kräfte und Momente der Flugmechaniksimulation bereitstellt.

Neben den Herausforderungen, die bei der Schaffung der Schnittstelle selbst bestanden, lag eine weitere Herausforderung auf Seiten der CFD-Simulation darin, die kommandier-

Abbildung 1.2: **CAD-Geometrie des FMTA inklusive Steuerflächenpatches und Box für modulare Netzgenerierung im Beschädigungsbereich (blauer Kasten).**



ten Steuerflächenausschläge während der Simulation in entsprechende Änderungen der FMTA-Geometrie umzusetzen. Ferner sollte im Rahmen des AP 5.4 des MiTraPor-II-Projekts die Funktion der entwickelten CFD-Simulink-Kopplung an einem ausgewählten Beschädigungsszenario des FMTA demonstriert werden.

1.4 Gliederung

Der vorliegende Bericht ist wie folgt gegliedert. Das nachfolgende Kapitel 2 erläutert, wie die Kopplung zwischen TAU und MATLAB/Simulink umgesetzt wurde. Die CFD-seitig implementierten Prozessteile werden in Kap. 2.1 (S. 15ff.) diskutiert. Der CFD-seitige Prozess basiert maßgeblich auf der Nutzung des *FlowSimulators*, welcher in Bezug auf seine Idee und grundsätzlichen Eigenschaften in Kap. 2.1.1 skizziert wird. Die Realisierung der Schnittstelle zwischen Python und Simulink wird in Kap. 2.2 (S. 26ff.) beschrieben. Die Kommunikation zwischen Simulink- und CFD-Seite fußt auf der Nutzung einer Socket-Kommunikationsschicht, die im Anhang C (S. 63ff.) erläutert wird.

In Kapitel 3 wird die konkrete Anwendung der TAU-Simulink-Kopplung auf die Simulation des FMTA mit einer exemplarischen Beschädigung demonstriert. Das im AP betrachtete Beschädigungsszenario und der verfolgte Ablauf zur Simulation dieses Szenarios werden in Kap. 3.1 vorgestellt. Kapitel 3.2 beschreibt die bei der Simulation verwendeten CFD-Netze. Wie die Vorgabe von Steuerflächenausschlägen beim konkreten Testfall realisiert wurde, wird in Kap. 3.3 demonstriert. Als Ausgangspunkt der Beschädigungssimulation muss der Trimmzustand des unbeschädigten FMTA bestimmt werden, der einem horizontalen unbeschleunigten Geradeausflug entspricht. Auf welche Weise dies mit der CFD-Simulink-Kopplung erfolgt ist, wird in Kap. 3.4 (S. 49ff.) erläutert. Im Anschluss an die Trimmsimulation wurde eine instationäre aerodynamisch-flugdynamisch gekoppelte Simulation für das beschädigte FMTA durchgeführt. Die Vorgehensweise und die erzielten Resulte werden in Kap. 3.5 (S. 50ff.) diskutiert.

Im Anhang ab S. 55 werden Detailinformationen gegeben, die im Rahmen der Simulationen des vorliegenden Berichts verwendet wurden. Die Zuordnung der TAU-Marker zu den Steuerflächen des FMTA und die Position der Scharnierachsen — beides wird zur Verstellung der Steuerflächen im Rahmen der CFD-Simulink-Kopplung benötigt — werden in Anhang A (S. 55ff.) aufgelistet. Die physikalischen Größen, die zwischen CFD- und Simulink-Simulation ausgetauscht werden, sind detailliert im Anhang B (S. 59ff.) angegeben. Alle im Rahmen der CFD-Simulink-Kopplung verwendeten Koordinatensysteme werden im Anhang E (S. 81ff.) vorgestellt. Im Anhang F (S. 85) wird die Software genannt, die für den Einsatz der in diesem Bericht beschriebenen Kopplung benötigt wird.

2 Realisiertes Konzept zur TAU-Simulink-Kopplung

Bei der Realisierung der Kopplungsschnittstelle besteht die in Tab. 2.1 zusammengefasste Ausgangssituation. Die berechnungsintensive CFD-Simulation (hier der TAU-Code) und der Simulink-Prozess laufen auf unterschiedlichen Rechnerarchitekturen unter unterschiedlichen Betriebssystemen. Die Simulink-Simulation ist ein serieller Prozess, der auf einem einzelnen Desktop-PC mit zumeist Windows-Betriebssystem läuft, (dies war hier der Fall) während TAU ein paralleler Prozess ist, der auf einer Vielzahl von Prozessoren auf einem Hochleistungsrechner (im vorliegenden Fall auf dem DLR-CASE-Cluster) unter einem Unix/Linux-Betriebssystem läuft. Der Simulink-PC ist in den meisten Fällen (so auch auf dem CASE-Cluster) nicht Teil des eigentlichen Cluster-Netzwerks. Die dezidierten Rechenknoten vieler Hochleistungsrechner haben zudem meist keine Verbindung zum Netzwerk *außerhalb* des Clusters — meist fungiert nur der Frontend-Knoten des Clusters als Bindeglied zwischen Rechenknoten und *Außenwelt*.

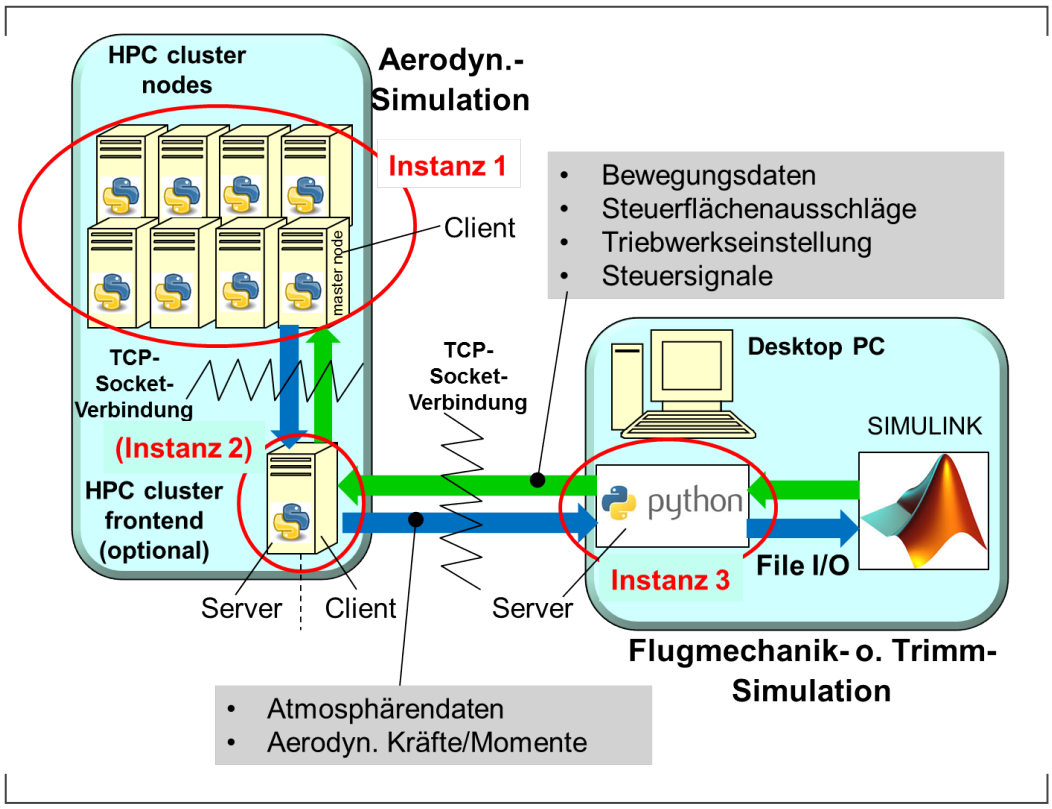
Die genannten Aspekte waren die maßgeblichen Gründe dafür, dass die Kopplungsschnittstelle in Form einer Netzwerk-Socket-Kommunikation realisiert wurde. Der konzeptionelle Aufbau der Schnittstelle ist grob in Abb. 2.1 illustriert und nachfolgend etwas detaillierter beschrieben.

Für den Fall, dass die Rechenknoten des Clusters keine Netzwerkverbindung *nach außen* besitzen, existieren drei Instanzen im realisierten Kopplungskonzept, andernfalls nur zwei. Die Instanzen sind in Abb. 2.1 rot umrandet. Jede Instanz wird durch ein Python-Prozessskript repräsentiert, das auf der jeweiligen Instanz gestartet wird (siehe z. B. die Skripte *run_CFD.py* (für Instanz 1) und *run_FM.py* (für Instanz 3) in Abb. 2.2 auf S. 14). Die Menge der Knoten auf denen die Berechnung der CFD-Simulation mit TAU läuft, bildet eine der drei Instanzen. Der Master-Knoten dieser Menge geht als Socket-Client eine Socket-Kommunikation mit der optionalen zweiten Instanz ein – dem Cluster-Frontend-Knoten. Letzterer tritt in diesem Fall gegenüber den Rechenknoten als SocketKommunikationsserver auf. Die dritte Instanz wird von der Simulink-Simulation gebildet bzw. der Python-Schicht, die die Simulink-Simulation umgiebt. Sie geht als Socket-Server wiederum

Tabelle 2.1: **Ausgangssituation/Rahmenbedingungen der TAU-Simulink-Kopplung**

Simulink	TAU
- läuft auf einzelnem Desktop-PC - meist unter Windows-Betriebssystem - *Unterbrechung* der Zeitintegration nicht vorgesehen. - kein Speicherzugriff von *außen* möglich[1]	- HPC-Cluster (massiv parallele Rechnung, verteilter Arbeitsspeicher und evtl. verteiltes Dateisystem) - meist UNIX-Betriebssystem - Rechnung unterliegt der zeitlichen Vorgabe des Scheduling-Systems - meist *kein* Zugang von Rechenknoten nach *außen* möglich

Abbildung 2.1: **Übersicht über das realisierte Konzept zur TAU-Simulink-Kopplung.**



eine Socket-Kommunikation mit dem Cluster-Frontend-Knoten ein. In diesem Fall bildet Letzterer einen Socket-Client gegenüber dem Simulink-PC. Der Cluster-Frontend-Knoten fungiert somit in dieser Konstellation lediglich als Mittelsmann ohne eigene Simulationsanteile — er empfängt lediglich Daten und reicht diese direkt an die nächste Instanz weiter. Zwischen dem eigentlichen Simulink-Prozess und der umgebenden Python-Schicht (Instanz 3) ist ein dateibasierter Datenaustausch realisiert (siehe Abb. 2.1). Dabei wurde in der Simulink-Simulation der Aerodynamikblock *abgeschaltet*, der das Derivativmodell enthält. Anschließend wurde in einen für externe Kräfte und Momente vorgesehenen Block eine Simulink-spezifische C++-S-Function implementiert. Weitergehende Details dazu werden in Kap. 2.2 erläutert.

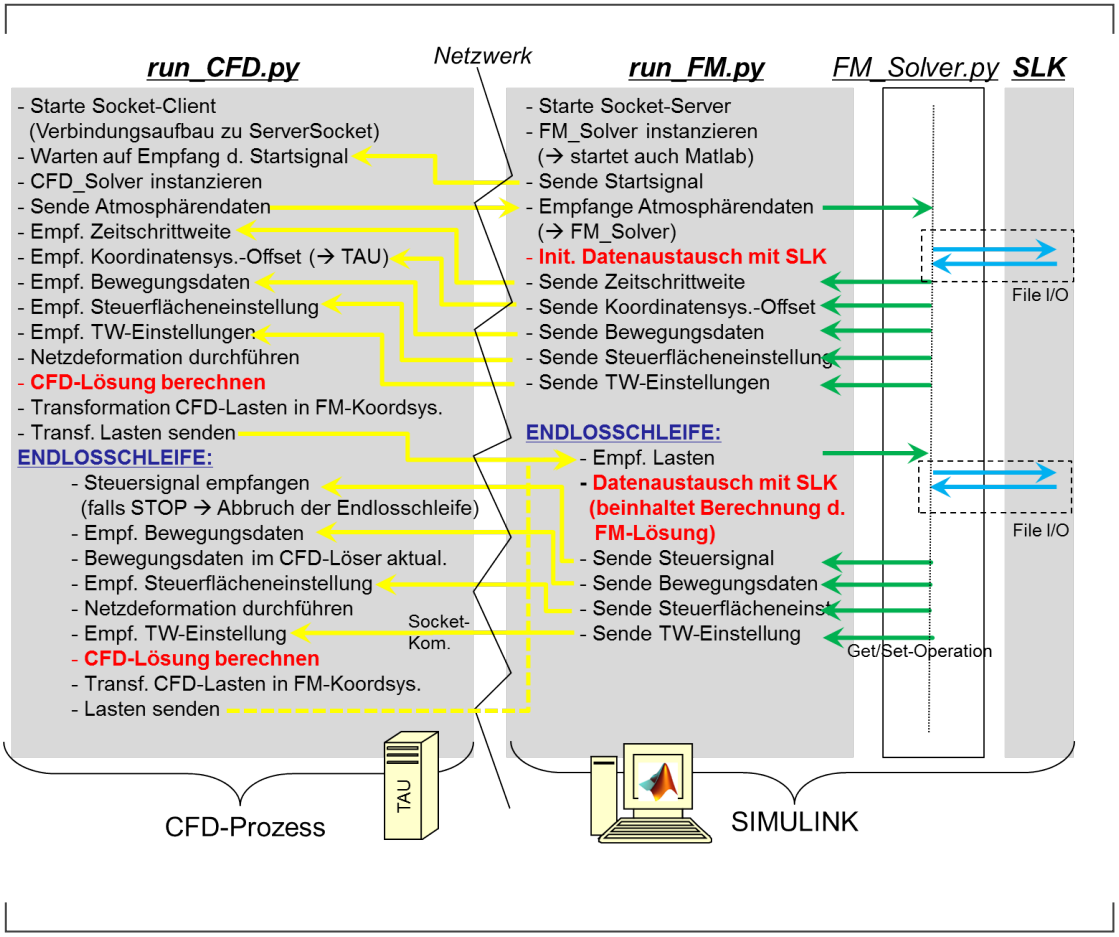
Folgende Daten werden über die Socket-Schnittstelle zwischen den Instanzen ausgetauscht. Zu Beginn der Simulation sendet die CFD-Simulation einmalig die Atmosphärendaten (Schallgeschwindigkeit, Luftdruck, Luftdichte und -temperatur). In jedem Kopplungsschritt sendet die CFD-Simulation (Instanz 1) die auf die simulierte Konfiguration einwirkenden aerodynamischen Kräfte und Momente. Von Seiten der Simulink-Simulation (Instanz 3) werden zu Beginn der Simulation einmalig die Zeitschrittweite[2] und der Verschiebungsvektor gesendet, der zwischen den CFD-seitigen und Simulink-seitigen Koordinatensystemen besteht. In jedem Kopplungsschritt werden ein Steuersignal (signalisiert Fortsetzung/Abbruch der Simulation), der Bewegungszustand (Position, Geschwindigkeit, Lagewinkel und Drehraten), die Ausschläge aller Steuerflächen (Querruder, Spoiler, Flaps, Trimmflosse, Höhenruder und Seitenruder) und die Schubeinstellungen aller vier Triebwerke von der Simulink-Simulation (Instanz 3) gesendet. Die Tabellen B.1 und B.2 (S. 59-61) listen detailliert auf, welche Größen ausgetauscht werden, deren Einheiten und in welchem Koordinatensystem die jeweilige andere Prozessseite die Größe erwartet.

Der Tatsache, dass die CFD-Simulation im parallelen Modus abläuft, muss auch die Socket-Kommunikationsschnittstelle (zumindest auf Seiten der CFD-Simulation) Rechnung tragen. Innerhalb der Schnittstelle werden dazu die Daten, die von der CFD-Simulation zu versenden sind, jeweils auf dem Master-Knoten gesammelt bzw. die Daten, die auf dem Master-Knoten von der Flugdynamiksimulation empfangen wurden, auf die Rechenknoten verteilt (für Details wird auf Abschnitt C verwiesen).

Abbildung 2.2 zeigt einen Ablaufplan der Kopplung zwischen TAU und Simulink, sowie der Datensende- und -empfangssequenzen. Die etwaig notwendige Kommunikation über den Cluster-Frontend-Knoten (3-Instanzen-Fall) ist in Abb. 2.2 aus Gründen der Übersichtlichkeit ausgespart. In Abb. 2.2 repräsentieren das Python-Skript *run_FM.py* und die Python-Klasse *FM_Solver* die "Instanz 3" aus Abb. 2.1, mittels denen die Interaktion mit dem eigentlichen Simulink-Prozess realisiert ist. Kap. 2.1.2 (S. 19ff.) und 2.2.1 (S. 26ff.) stellen

[2] In CFD- und Flugmechaniksimulation wird derzeit dieselbe einheitliche Zeitschrittweite verwendet.

Abbildung 2.2: **Prinzipielle Ablaufsequenz der TAU-Simulink-Kopplung. *Legende*: ← Sende- bwz. Empfangsprozess über Socket-Kommunikation. →/← Setzen/Abfragen einer Membervariable. →/← Schreiben/Lesen aus Datei.**



die beiden Prozessskripte *run_CFD.py* und *run_FM.py* etwas detaillierter vor.

Im Fall der CFD-Simulink-Kopplung im Flugdynamiksimulationsmodus ist derzeit ein sogenanntes loses Kopplungsschema realisiert, d. h. es erfolgt stets nur ein einmaliger Austausch zwischen CFD- und Flugmechaniksimulation pro Zeitschritt (wie in Abb. 2.2 gezeigt). Die entwickelten Python-Prozessskripte können ohne Veränderung gleichermaßen im Trimm- und Flugdynamiksimulationsmodus eingesetzt werden.

2.1 CFD-seitiger Prozessteil

Der CFD-seitige Prozessteil beruht maßgeblich auf der Nutzung der *FlowSimulator*-Softwareumgebung. Bevor daher Details des eigentlichen Prozessskripts *run_FM.py* erörtet werden, werden nachfolgend einige wenige wesentliche Aspekte dieser Software beschrieben.

2.1.1 FlowSimulator

Der *FlowSimulator* ist eine gemeinsam von Airbus, Airbus Defence & Space, DLR, ONERA und Universitäten entwickelte Softwareumgebung. Die Entwicklung ziehlt insbesondere auf den Einsatz in massiv parallelen multi-disziplinären Simulationen im Aerodynamik-Kontext ab, d. h. der Kopplung von CFD mit anderen Disziplinen.

Der grundsätzliche Aufbau des *FlowSimulators* ist in Abb. 2.3 illustriert. Man kann sich den *FlowSimulator* schichtenweise organisiert vorstellen. Den untersten und wichtigsten Baustein des FlowSimulators bildet der *FSDataManager*, kurz *FSDM*. Gemäß der Grundphilosophie des FlowSimulator dürfen Daten einzelner Prozessbausteine nur über den Weg durch den *FSDM* und nicht direkt miteinander ausgetauscht werden. Dies ist exemplarisch für einen generischen Prozess in Abb. 2.4 veranschaulicht. TAU importiert darin das partitionierte CFD-Netz und eine etwaig vorhandene Lösung aus dem *FSDM*, berechnet eine Lösung, und exportiert diese zurück zum *FSDM*. Nachgeschaltete Prozesskomponenten importieren jeweils wieder die Daten aus dem FSDM und operieren auf diesen. Es gibt also keine "horizontalen" Datenströme, sondern nur "vertikale". So bleibt stets die Lauffähigkeit des Gesamtprozesses unabhängig von der Existenz einzelner Prozesskomponenten gewahrt. Denn es bestehen keine Abhängigkeiten der Prozesskomponenten untereinander.

Abbildung 2.3: **Aufbau der *FlowSimulator*-Softwareumgebung. Der *FlowSimulator Data Manager* (*FSDM*) repräsentiert das Rückrat des *FlowSimulator*.**

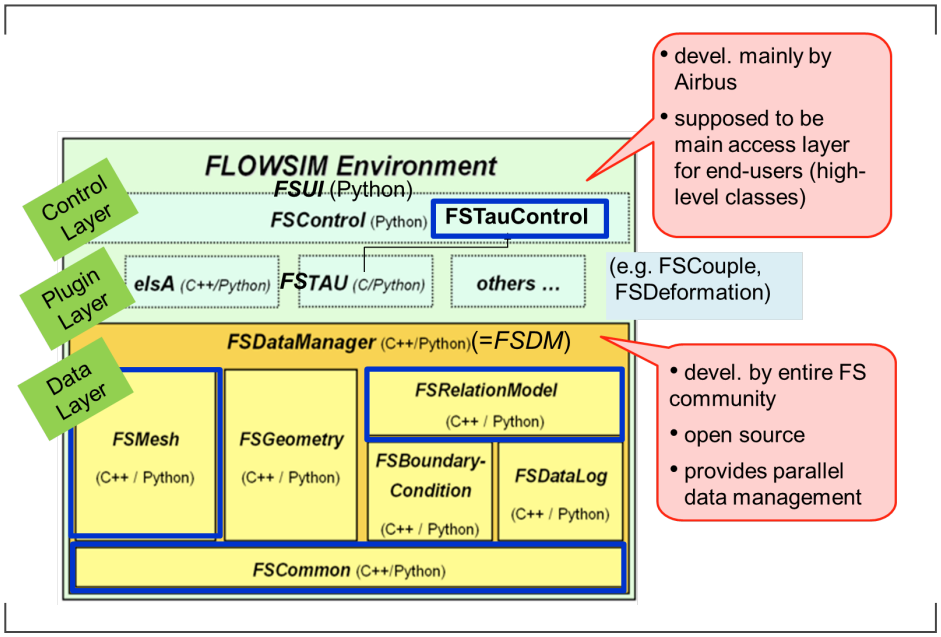
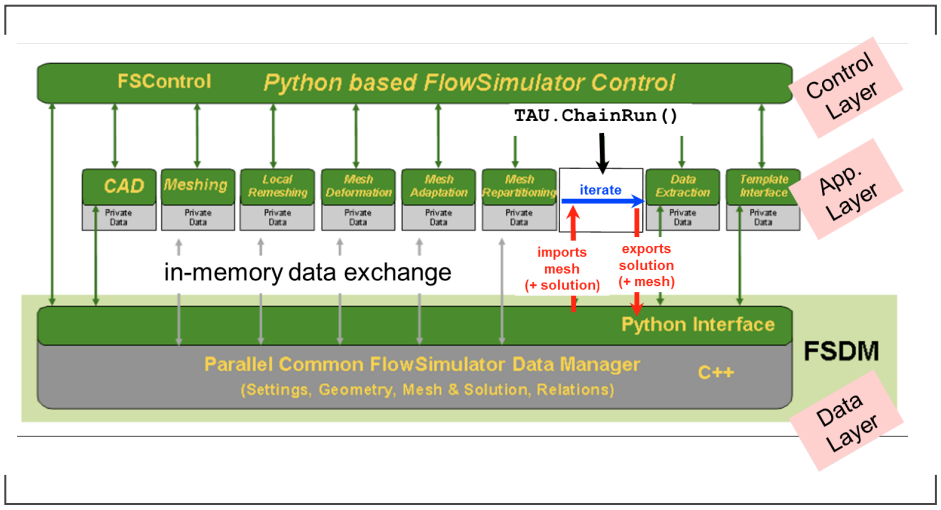


Abbildung 2.4: **Idealisierter *FlowSimulator*-Prozess. Der Prozess zeigt das fundamentale Entwicklungskonzept für *FlowSimulator*-Prozessketten — der Datenaustausch der Prozesskomponenten findet lediglich über den *FSDM* statt (s. Abb. 2.3), anstelle eines direkten Datenaustauschs zwischen Prozessketten.**



Einige wesentliche Bestandteile des *FSDM* sind in Abb. 2.3 dargestellt. Der *FSDM* ist eine Ansammlung von Datentypen, die aus Effizienzgründen in C++ implementiert sind. Die jeweiligen Datentypen fungieren als zentrale Schnittstelle für einen etwaigen Datenaustausch mit Anwendungen (den sogenannten *Plugins*), wie z. B. den CFD-Löser TAU. Die wichtigsten Datentypen sind in der *FSMesh*-Bibliothek enthalten. Sie stellt eine Implementierung für die Behandlung verteilter mehrblock-strukturierter und unstrukturierter CFD-Netze bereit. Für die *FSMesh*-Klasse sind Methoden zum Importieren/Exportieren des Netzes in verschiedene Datenformate (z. B. TAU-NetCDF, HDF5, TECPLOT, CGNS, etc.) implementiert, sowie verschiedene Partitionierer zur Gebietszerlegung des Netzes (z. B. geometrisch orientierte Verfahren wie RCB[3] und graphen-basierte Verfahren wie ParMetis[4] und Zoltan[5]).

Ein weitere Bestandteil des *FSDM* ist das *FSRelationsModel*. Es fungiert im wesentlichen als Datenbank für Simulationsparameter, die zwischen verschiedenen Plugins ausgetauscht werden müssen, und es dient als Ablage für mit Netzrändern assoziierten Randbedingungen. Einen Auszug aus einem FSRelationsModel, das als XML-Datei exportiert wurde, zeigt Programmausdruck 2.1.

Programmausdruck 2.1: **Beispiel für ein *FSRelationsModel* (exportiert als XML-Datei).**

```
<?xml version="1.0" encoding="UTF-8"?>
<FS:RelationsModel xmlns:FS="http://www.eads.com/FLOWSIM/1.0/">
  <FS:ModelTree>
    <FS:Node Name="/">
      <FS:Node Name="/CFDDomain">
        <FS:AttributeList>
          <FS:Attribute Name="MeshKey">cfdMesh</FS:Attribute>
        </FS:AttributeList>
        <FS:Node Name="/CFDDomain/Aileron">
          <FS:StringAssignment Type="CADGroupName">
            <FS:String>Aileron</FS:String>
          </FS:StringAssignment>
          <FS:IDAssignment Type="CADGroupID">
            <FS:IDList>22</FS:IDList>
          </FS:IDAssignment>
          <FS:BCAssignment Type="CFD">
            <FS:BCEulerWall>
              <FS:AttributeList>
                <FS:Attribute Name="BCName">BCEulerWall</FS:Attribute>
                <FS:Attribute Name="CENTAUR.BoundaryCondition">1020</FS:Attribute>
              </FS:AttributeList>
            </FS:BCEulerWall>
          </FS:BCAssignment>
        </FS:Node>
```

[3] *Recursive Coordinate Bisection*
[4] ParMetis: **glaros.dtc.umn.edu/gkhome/metis/parmetis/overview**
[5] Zoltan: **www.cs.sandia.gov/zoltan**

Darin ist ein Oberflächenanteil, der einen Teil des Querruders repräsentiert, mit einer Euler-Randbedingung versehen. Dieser Oberflächenanteil kann über das Attribut *CADGroupName* namentlich im RelationsModel angesprochen werden (hier: "Aileron"), oder über das Attribut *CADGroupID* über eine eindeutige Nummer identifiziert werden (hier: "22").

Die oberste Ebene des *FlowSimulators* — die sogenannte Kontrollschicht — wird von Prozessskripten gebildet. Die Prozessskripte sind im Sinne einer möglichst schnellen Erstellung von multi-disziplinären Prozessketten in Python implementiert. Die Bestandteile des *FSDM* wurden zur Nutzung in den Prozessskripten mittels SWIG nach Python "gewrappt".

Programmausdruck 2.2 gibt ein rudimentäres Beispiel für ein FlowSimulator-Prozessskript. In diesem wird ein TAU-Netz und eine dazugehörende Lösung importiert, das Netz anschließend mit RCB partitioniert, und ein *FSRelationsModel* im *FSDM* aus den TAU-Markerinformationen im Netz initialisiert. Entgegen der üblichen Vorgehensweise, bei der direkt ein FSMesh-Objekt instanziert wird, wird in Programmausdruck 2.2 zunächst eine Instanz des FSDataManagers erzeugt. Anschließend wird von diesem ein leeres *FSMesh*-Objekt entgegengenommen, das fortan unter dem Schlüssel "CFDMesh" im *FSDM* registriert und ansprechbar ist. Der Grund für diese Vorgehensweise ist, dass die *FlowSimulator*-Methode zur Verstellung der Steuerflächen mittels Netzdeformation (*FSCtrlSurfDisplFieldGenerator*[6]), die hier im Rahmen der Kopplung verwendet wurde, eine Instanz des *FSDataManagers* mit initialisiertem *FSRelationsModel* benötigt.

Programmausdruck 2.2: **Beispiel für die Importierung eines TAU-Netzes (Datei `tau.grid`) und einer TAU-Lösung (Datei `tau.pval`) und anschließender Partitionierung mit RCB im *FlowSimulator*.**

```
1  import FSDM
2  from FSDataManager import FSDataManager, FSClac

4  Clac = FSClac()
   DM = FSDataManager(Clac)
6  CFDMesh = DM.GetMesh("CFDMesh")

8  CFDMesh.ImportMeshTAU(MeshFilename="tau.grid")
   CFDMesh.ImportMeshTAUng(DataFilename="tau.pval")
10 CFDMesh.RepartitionMeshRCB()

12 DM.InitRelationsModelFromMesh("CFDMesh")
```

Es existiert eine Sammlung von Beispielskripten, die die Nutzung einzelner Bestandteile

[6] *FSCtrlSurfDisplFieldGenerator* ist eine *FlowSimulator*-Anwendung zur Verstellung von Steuerflächen mittels Netzdeformation. Sie ist Bestandteil des SVN-Repositories von *FSDeformation*. Sie ist verfügbar unter **`https://dev2.as.dlr.de/svn/fsdeformation/trunk/FSCtrlSurfDisplFieldGenerator`**

des *FSDM* demonstrieren[7]. Auf diese Skripte muss an dieser Stelle für weitere Details zum Funktionsumfang des *FSDM* verwiesen werden.

Neben der Anbindung von TAU an den *FSDM* (Plugins *FSTau* und *FSTauInterface*) werden darüberhinaus die folgenden *FlowSimulator*-Plugins im Rahmen der CFD-Simulink-Kopplung verwendet:

***FSDeformation* (inkl. *FSWallDistance*):** Methode im *FlowSimulator* zur Deformation von CFD-Volumennetzen mittels RBF[8]-basierter *Scattered Data Interpolation* [4–6]. Die Wirkung der berechneten Oberflächen-*Splines* auf jeden CFD-Punkt — jeder Spline repräsentiert ein Funktional für ein vorgegebenes Oberflächenverschiebungsfeld — wird mit den jeweilgen Wandabständen gewichtet, die von *FSWallDistance* berechnet werden. Das SVN-Repository des Plugins ist `https://dev2.as.dlr.de/svn/fsdeformation`.

***FSNumPyInterface*:** Dient zur Konvertierung zwischen dem *FSDM*-eigenen mehrdimensionalen Datenobjekt *FSArray* und *NumPy-Arrays*, auf die orginäre *NumPy*-Operationen anwendbar sind. Das SVN-Repository des Plugins ist `https://dev2.as.dlr.de/svn/fsnumpy`.

2.1.2 Prozesssteuerungsskript *run_CFD.py*

Jeder auf dem Cluster gestartete CFD-Prozess führt im Rahmen der CFD-Simulink-Kopplung das Prozessskript *run_CFD.py* aus. Gemäß der in Abb. 2.2 (S. 14) gezeigten prinzipiellen Ablaufsequenz lassen sich im wesentlichen die folgenden Aktionen im Skript identifizieren:

- Etablierung der Socket-Kommunikation
- Sende- und Empfangsoperation über die Socket-Schnittstelle
- Verstellung der Steuerflächen
- CFD-Berechnung

Nachfolgend werden diese Teilbereiche des Prozessskripts gesondert betrachtet.

[7] *FSDM*-Beispielanwendungsskripte sind unter `https://dev2.as.dlr.de/svn/fsdm/trunk/example` verfügbar
[8] RBF: Radiale Basisfunktion

Etablierung der Socket-Kommunikation

Im Rahmen der CFD-Simulink-Kopplung wurde die Python-Klasse *Parallel_ClientSocket_Iface* implementiert, die dazu dient, eine Socket-Kommunikationsverbindung im parallelen Kontext als Client aufzubauen. Der Konstruktor von *Parallel_ClientSocket_Iface* benötigt eine Instanz der Klasse *FSClac*. Letztere ist die Klasse des *FSDM* zur Handhabung paralleler Kommunikation im *FlowSimulator*-Kontext. Weitere Details dieser Klasse werden in Anhang C (ab S. 63ff.) beschrieben.

Programmausdruck 2.3 zeigt, wie der Socket-Kommunikationsclient basierend auf den Angaben zu IP-Adresse und Port des Socket-Server eine Verbindung zu Letzterem aufbaut. In Zeile 12 des Listings 2.3 wird eine Instanz des Clients erzeugt (Objekt *CFD_FM_Socket_Iface*) und in der darauffolgenden Zeile die eigentliche Verbindung mit der Methode *CreateConnection* erstellt. Danach verweilt der CFD-seitige Prozess in einer Endlosschleife bis vom Simulink-seitigen Prozess ein Startsignal (hier: der Wert *START_SIGNAL*=1) gesendet wird.

Programmausdruck 2.3: **Instanzierung eines im parallelen Kontext funktionsfähigen Clients für die Socket-Kommunikation.**

```
1  import FSDM
2  from FSDataManager import FSLog, FSClac
   from Parallel_ClientSocket_Iface import Parallel_ClientSocket_Iface
4  import time

6  Clac = FSClac()

8  # Angabe des Ports und der IP-Adresse der Servers der Socket-Kommunikation
   # mit dem die Verbindung erfolgen soll
10 portSlkMachine = 65534
   ipAddressSlkMachine = '10.148.2.81'
12 CFD_FM_Socket_Iface = Parallel_ClientSocket_Iface(Clac, ipAddressSlkMachine,
       portSlkMachine)
13 CFD_FM_Socket_Iface.CreateConnection()
14
   # Wait until start signal is given by Simulink process
16 while True:
      FSLog(Clac, 0, "Waiting for START signal\n")
18    signal = CFD_FM_Socket_Iface.Receive()
      if signal == CFD_FM_Socket_Iface.START_SIGNAL:
20       break
      time.sleep(5) # wait 5 seconds
22 Clac.Barrier()
   FSLog(Clac, 0, "Received START signal")
```

Sende- und Empfangsoperation über die Socket-Schnittstelle

Programmausdruck 2.4 zeigt beispielhaft die Nutzung des parallelen Socket-Kommunikationsclients zum Versenden und Empfangen von Daten. Der Reihe nach werden zuerst die Atmosphärendaten gesendet[9], anschließend die Zeitschrittweite der Simulation, der Versatz zwischen den Koordinatensystemen der CFD- und Simulink-Simulation (siehe Koordinatensystem im Anhang E), die Bewegungsdaten (Z. 9), die Steuerflächenstellungen (Z. 14) und als letztes die Triebwerkseinstellungen (Z. 20). Gesendet wird mittels der Methode *Send* und Empfangen mit *Receive*. Beim Versenden warten alle Prozesse in einer Kommunikationsbarriere bis der Master-Prozess den Versandt abgeschlossen hat. Nach dem Empfang der Daten *broadcasted* der Master-Prozess diese an alle CFD-Prozesse. Die Daten werden jeweils als der Datentyp empfangen, als der sie auch gesendet wurden. In den meisten Fällen werden orginäre Python-Listen versendet. Für Details wird auf Anhang C verwiesen (S. 63ff.).

Programmausdruck 2.4: **Exemplarischer Versende- und Empfangsprozess auf CFD-Seite.**

```
1  import numpy as np
   import string
3
   CFD_FM_Socket_Iface.Send(atmosphere) # Atmospaerendaten empf.
5
   deltaT = CFD_FM_Socket_Iface.Receive() # Zeitschrittweite empf.
7  cosOffset = CFD_FM_Socket_Iface.Receive() # Koordinatensystemversatz empf.

9  motionList = CFD_FM_Socket_Iface.Receive() # Bewegungsdaten empf.
10 motionString = string.join(map(str,motionList))
11 pb, qb, rb, phi, theta, psi, ug, vg, wg, xg, yg, zg = motionList
   # ... danach Verwendung von pb, qb, rb, etc. zur Vorgabe der Bewegung in TAU
13
   ctrlSurfSettingsRad = CFD_FM_Socket_Iface.Receive() # Steuerflaechenstellungen empf.
15 ctrlSurfSettingsDeg = map(np.rad2deg, ctrlSurfSettingsRad)
   (aileronLeftDeflAngle, aileronRightDeflAngle, elevatorLeftDeflAngle,
       elevatorRightDeflAngle, rudderDeflAngle, htpDeflAngle, flapsDeflAngle,
       rightSpoiler1DeflAngle, rightSpoiler2DeflAngle, rightSpoiler3DeflAngle,
       rightSpoiler4DeflAngle, rightSpoiler5DeflAngle, leftSpoiler1DeflAngle,
       leftSpoiler2DeflAngle, leftSpoiler3DeflAngle, leftSpoiler4DeflAngle,
       leftSpoiler5DeflAngle) = ctrlSurfSettingsDeg
17 # ... danach Verwendung von  aileronLeftDeflAngle, etc. zur Verstellung
   #     der Steuerflaechen
19
   engineSettings = CFD_FM_Socket_Iface.Receive() # Triebwerkseinstellung empf.
21 (thrustLeftOutboardEngine, thrustLeftInboardEngine, thrustRightInboardEngine,
       thrustRightOutboardEngine) = engineSettings
```

[9] Die Atmosphärendaten werden zu Beginn der Simulation von TAU abgefragt, an Simulink übermittelt und bleiben konstant über die gesamte Simulationsdauer — unabhängig davon, ob sich die Flughöhe des FMTA ändert.

Verstellung der Steuerflächen

Für diesen Aspekt wird auf Kap. 3.3 (S. 38ff.) verwiesen. Dort werden die Anteile des Prozessskripts *run_CFD.py*, die eigens der Verstellung der Steuerflächen zugeordnet sind, direkt am Beispiel des FMTA beschrieben.

CFD-Berechnung

Die TAU-Simulink-Kopplung nutzt die Anbindung von TAU an Python — genannt TAU-Python [8]. Zur leichteren Integration von TAU in die *FlowSimulator*-basierte CFD-Simulink-Prozesskette wurde eine zusätzliche Python-Schicht um TAU-Python herum erstellt. Zur bequemen Steuerung des Lösers im FlowSimulator-Kontext wurde die Klasse *TauControl* implementiert. Sie ist adaptiert von der *TauControl*-Implementierung im *FlowSimulator*-Plugin *FSControl*, das von Airbus entwickelt wird, aber im Rahmen des Projekts nicht eingesetzt werden konnte.

Der Programmausdruck 2.5 zeigt auszugsweise das Python-Skript *run_CFD.py* — insbesondere die darin implementierte Zeitschleife. In Zeile 15 erfolgt die Instanzierung von *TauControl*. Als Argumente erhält der Konstruktur das *FSClac*-Objekt, das *FSMesh*-Objekt, auf dem die CFD-Berechnung erfolgen soll, den TAU-spezifischen Namen des Lösertyps und den Namen der TAU-Parameterdatei. Innerhalb des Konstruktors erfolgt dann die Importierung aller wesentlichen TAU-Python-Module[10] aus dem Verzeichnis, das unter der Umgebungsvariable *FSTAU_PYTHON* gesetzt ist. Zudem wird die parallelen Umgebung von TAU im Konstruktor initialisiert.

Programmausdruck 2.5: **Auszug aus *run_CFD.py*. Fokus auf Vorgabe der von Simulink gesendeten Bewegungsdaten an TAU und Implementierung der CFD-seitigen Zeitschleife.**

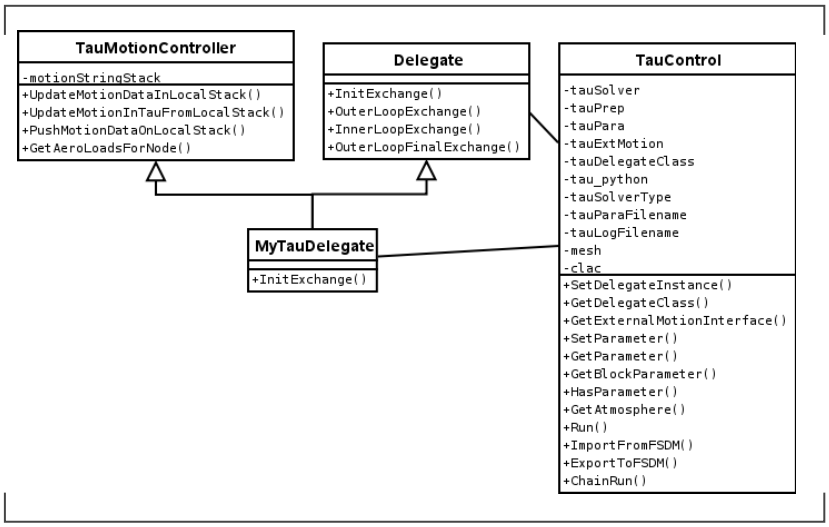
```
1  import FSDM
   from FSDataManager import FSClac, FSMesh
3
   from TauControl import TauControl
5  from TauMotionController import TauMotionController

7  Clac = FSClac()
   CFDMesh = FSMesh(Clac)
9
   # Importierung des CFD-Netzes
11 # Initiale Send- und Empfangsoperationen
   # Initiale Verstellung der Steuerflaechen
```

[10] *PySolv*, *PyPrep*, *PyPara*, *PyFsdmTauInterface*, *PyMotionExternal* und *PyDelegate*

```
13 # (Aktionen hier ausgespart !)
14
15 Tau = TauControl(Clac, CFDMesh, "py_el", "tau.para")
16 TauDelegateClass = Tau.GetDelegateClass()
17 TauExtMotionInterface = Tau.GetExternalMotionInterface()
18
19 # Definition der Klasse, die die externe Bewegungsvorgabe mit TAU durchfuehrt
   # (Die Methode "InitExchange" wird von TAU
21 #  vor der Berechnung der CFD-Loesung aufgerufen)
   class MyTauDelegate(TauMotionController, TauDelegateClass):
23     def __init__(self, Clac, TauExtMotionInterface):
           TauMotionController.__init__(self, Clac, TauExtMotionInterface)
25
       def InitExchange(self):
27         FSLog(self.GetClac(), 0, "Performing initial motion exchange")
           self.UpdateMotionInTauFromLocalStack()
29         self.PushMotionDataOnLocalStack()
30
31 MyTauDelegate = MyTauDelegate(Clac, TauExtMotionInterface)
32 # Ersetzen der "funktionsleeren" Delegate-Instanz in TauControl durch die vom Nutzer
       implementierte
33 # abgeleitete Klasse
   Tau.SetDelegateInstance(MyTauDelegate)
35
36 # Zeitschleife
37 while True:
38
39     # Empf. etwaiges Abbruchsignal von Simulink-Seite
       signal = CFD_FM_Socket_Iface.Receive()
41     if signal == CFD_FM_Socket_Iface.STOP_SIGNAL:
           break
43
44     # Empf. Bewegungsdaten von Simulink-Seite
45     motionList = CFD_FM_SocketIface.Receive()
46     motionString = string.join(map(str,motionList))
47
48     # Setze Bewegungsdaten im Verwaltungsstack
49     MyTauDelegate.UpdateMotionDataInLocalStack("FMTA", motionString)
50
51     # Empf. weiterer Daten (Steuerflaechen-, Triebwerkseinstellung)
       # Fuehre Netzdeformation zur Steuerflaechenverstellung aus
53     # (Aktionen hier ausgespart !)
54
55     # Importiere CFD-Netz und Loesung vom FSDM nach TAU
       Tau.ImportFromFSDM()
57
58     # Berechne TAU-Loesung eines Zeitschritts
59     Tau.Run()
60
61     # Hole aerodynamische Lasten fuer Bewegungsknoten mit Namem "FMTA" aus TAU-Speicher
       loads = MyTauDelegate.GetAeroLoadsForNode("FMTA")
63
64     # Uebertrage Netz und Loesung von TAU zurueck zum FSDM
65     Tau.ExportToFSDM()
66
```

Abbildung 2.5: **UML-Diagramm der Beziehungen zwischen den Klassen *TauMotionController*, *TauControl*, *Delegate* und *MyTauDelegate*, die zur Steuerung von TAU auf *FlowSimulator*-Ebene und zur Vorgabe der Bewegung an TAU verwendet werden.**




```
67      # Sende Lasten zur Simulink-Seite
        CFD_FM_Socket_Iface.Send(loads)
69  # Ende der Zeitschleife
```

Die Vorgabe der Bewegung des FMTA für die CFD-Rechnung mit TAU wurde in der folgenden Weise realisiert. TAU bietet über die *Delegate*-Klasse des TAU-Python-Moduls *PyDelegate* die Funktionalität, an dezidierten Interaktionspunkten im TAU-Python-Ablauf einen Datenaustausch mit externen Lösern durchzuführen bzw. Bewegungsdaten vorzugeben. Gemäß diesem Konzept ist eine eigene Klasse von der *Delegate*-Klasse abzuleiten, für die die in der *Delegate*-Klasse vorbereiteten Schnittstellenmethoden implementiert werden müssen. Infolge der hier verfolgten losen Kopplung zwischen TAU und Simulink ist lediglich die Methode *InitExchange* zu implementieren. Denn der TAU-Löser wird hier nach jedem Zeitschritt *gestoppt* und zu Beginn jedes Zeitschritts neu basierend auf den gegebenen Bewegungsdaten initialisiert. *InitExchange* ist im Rahmen der vom Nutzer implementierten Klasse *MyTauDelegate* implementiert (siehe Z. 22-29 des Programmausdrucks 2.5). *MyTauDelegate* ist abgeleitet von den Klassen *Delegate* und *TauMotionController* (siehe UML-Diagramm in Abb. 2.5). Letzterer wird als Programmausdruck D.6 (S. 78) im Anhang gezeigt. Der *TauMotionController* verwaltet einen lokalen *Stack* der an TAU zu übertragenden Bewegungsdaten. In TAU werden drei Zeitebenen im Rahmen der instationären Zeitintegration mit 2. Ordnung genutzt: $\mathcal{M}^{n+1}$, $\mathcal{M}^{n}$, $\mathcal{M}^{n-1}$, mit $\mathcal{M}$ als Spaltenmatrix, die die Bewegungsdaten enthält, und n als dem Index der Zeitebene. $n+1$ repräsentiert die

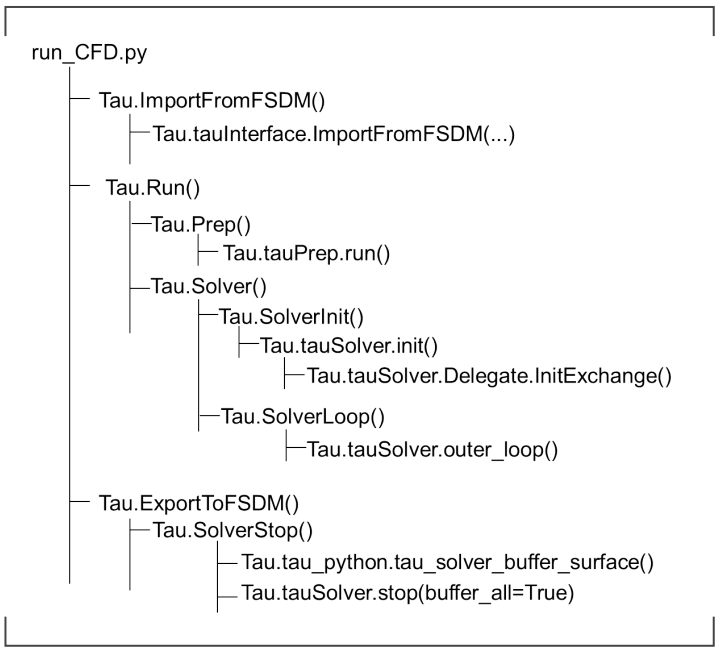
aktuellste Zeitebene, für die in TAU die Berechnung der CFD-Lösung erfolgt. Die Lösung der *alten* Zeitebenen gehen im Rahmen der verwendeten Dual-Time-Stepping-Methode zur Zeitintegration auf der *rechte Seite* ein. Für alle drei Zeitebenen müssen jeweils Bewegungsdaten vorgegeben werden. Der *Stack* in *TauMotionController* hält genau diese drei Zeitebenen vor. Mit der Methode *UpdateMotionDataInLocalStack* in Zeile 49 des Programmausdrucks D.6 wird der *oberste* Bewegungszustand $\mathcal{M}^{n+1}$ auf dem *Stack* des *TauMotionController* mit den aktuell von Simulink empfangenen Bewegungszustand $\mathcal{M}^{SLK}$ überschrieben: $\mathcal{M}^{n+1} \leftarrow \mathcal{M}^{SLK}$. Die Bewegungsdaten befinden sich zu diesem Zeitpunkt allerdings weiterhin auf dem *Stack* und wurden noch nicht an TAU übertragen.

In Zeile 59 wird die eigentliche Berechnung der CFD-Lösung mit TAU ausgeführt. Bevor die eigentlichen TAU-Iterationen durchgeführt werden, wird die Methode *UpdateMotionInTauFromLocalStack* von *MyTauDelegate* innerhalb von *TauControl* aufgerufen (Z. 28). Innerhalb dieser werden die Bewegungsdaten, die sich für alle drei betrachteten Zeitebenen derzeit auf dem *Stack* des *TauMotionController*s befinden, an TAU übertragen. Basierend auf den übertragenen Bewegungsdaten erfolgt dann die Berechnung eines Zeitschritts mit der Dual-Time-Stepping-Methode in TAU. In *InitExchange* werden ferner die Daten im *Stack* des *TauMotionController*s danach innerhalb der Methode *PushMotionDataOnLocalStack* (Z. 29) eine Zeitebene weiter nach *unten* geschoben, d. h. $\mathcal{M}^{n-1} \leftarrow \mathcal{M}^{n}$ und $\mathcal{M}^{n} \leftarrow \mathcal{M}^{n+1}$. Dies gewährleistet, dass im darauf folgenden Zeitschritt die Bewegungsdaten für die Zeitebene $n+1$ wieder mit den Simulink-Daten aktualisiert werden können (d. h. $\mathcal{M}^{n+1} \leftarrow \mathcal{M}^{SLK}$) und dass jeweils die korrekte zeitliche Historie der Bewegung an TAU übermittelt wird.

Der Austausch mit dem *FlowSimulator* bzw. dem *FSDM* erfolgt innerhalb der Zeitschleife in den Zeilen 56 und 65. Mittels *ImportFromFSDM* wird jeweils das aktuelle CFD-Netz, das die Steuerflächenverstellung als Deformation enthält, und die aktuelle CFD-Lösung vom *FSDM* an TAU übertragen. Nach erfolgter CFD-Berechnung werden Lösung und Netz wieder mit der Methode *ExportToFSDM* an den *FSDM* zurückgesendet. Dieser Schritt gibt den internen TAU-Speicher wieder frei. Daher ist es wesentlich, dass zuvor die integralen aerodynamischen Lasten von TAU an die Prozessskriptebene übertragen werden (siehe Z. 62). Diese werden zum Ende der Zeitschleife an Simulink gesendet (Z. 68), so dass mit diesen innerhalb von Simulink die flugdynamische Simulation für einen Zeitschritt oder eine Trimmiteration erfolgen kann.

Abbildung 2.6 dient abschließend dazu, die Interaktion von *TauControl* mit den nativen TAU-Python-Klassen zu veranschaulichen. Die Abbildung zeigt die Abfolge der Methodenaufrufe innerhalb von *TauControl*.

Abbildung 2.6: **Ablauf der Methodenaufrufe innerhalb der Methoden von *TauControl*.**



2.2 Simulink-Python-Schnittstelle

2.2.1 Prozesssteuerungsskript *run_FM.py*

Auf dem einzelnen Desktop-Computer, auf dem die flugdynamische Reaktion des FMTA-Modells in Simulink berechnet werden soll, wird das Python-Prozessskript *run_FM.py* gestartet. Es stellt gemäß Abb. C eine Verbindung über eine Socket-Schnittstelle zwischen dem Simulink-Prozess und dem CFD-Prozess her, d. h. entweder zum optionalen Sende- und Empfangsagenten auf dem Cluster-Frontend oder zum Master-Rechenknoten. Der Ablauf von *run_FM.py* ist prinzipiell im rechten Teil der Abb. 2.7 skizziert. Er soll im Folgenden etwas detaillierter erläutert werden — insbesondere die Interaktion mit MATLAB/Simulink.

Im ersten Schritt wird von *run_FM.py* ein Serverdients für die Socket-Kommunikation gestartet, der auf Verbindungsanfragen von Clients (in diesem Fall des HPC-Clusters) wartet (für Details wird wieder auf Anhang C verwiesen). Ist eine Verbindung hergestellt, so wird eine Instanz der Klasse *FM_Solver* erzeugt und das Signal zum Starten der Rechnung an die CFD-Prozesse gesendet. Der Programmausdruck 2.6 zeigt diese ersten Aktionen des Prozessskripts *run_FM.py*.

Programmausdruck 2.6: **Etablierung einer Socket-Verbindung zur CFD-Seite und Instanzierung der Wrapper-Klasse *FM_Solver* für die Kommunikation zwischen Python und Simulink.**

```
ipAddressServer="" # localhost
portServer=50002

# Erstellung eines Socket-Servers
CFD_FM_Socket_Iface = ServerSocket_Iface(ipAddressServer, portServer)
CFD_FM_Socket_Iface.CreateConnection()

# Instanzierung des Simulink-Wrappers
matlabExeCommand = "matlab"
slkControlFilePrefix = "FMTA_Tau_trim"
FM_Solver = FM_Solver(slkControlFilePrefix, matlabExeCommand=matlabExeCommand)

# Senden des Start-Signals an CFD-Seite
CFD_FM_Socket_Iface.Send(FM_Solver.START_SIGNAL)
```

Die Klasse *FM_Solver* dient als Zwischenschicht zwischen der Python-Welt und der Simulink-Welt. In *FM_Solver* findet der datei-basierte Datenaustausch mit Simulink statt und die von Simulink empfangenen Daten werden in Membervariablen der Klasse gespeichert, die dann von der Python-Seite aus zu gegebener Zeit abgefragt werden können, d. h. vor dem Moment der Übertragung der Daten über die Socketschnittstelle an die CFD-Seite. Dies zeigt der Programmausdruck 2.7, der den initialen Datenaustausch und die eigentliche Zeitschleife auf Seiten der flugdynamischen Simulation darstellt.

Programmausdruck 2.7: **Etablierung einer Socket-Verbindung zur CFD-Seite und Instanzierung einer Wrapper-Klasse für die Kommunikation zwischen Python und Simulink.**

```
# Empf. Atmosphaerendaten von CFD-Seite
atmosphere = CFD_FM_Socket_Iface.Receive()
FM_Solver.SetAtmosphere( atmosphere )

# Fuehre initialen Austausch der Daten zwischen FM_Solver
# und Simulink aus
FM_Solver.ExchangeInitData()

# Sende Daten an CFD-Seite
deltaT = FM_Solver.GetTimestepSize()
CFD_FM_Socket_Iface.Send( deltaT )
cosOffset = FM_Solver.GetCoordinateSystemOffset()
CFD_FM_Socket_Iface.Send( cosOffset )
motion = FM_Solver.GetMotion()
CFD_FM_Socket_Iface.Send( motion )
ctrlSurfSettings = FM_Solver.GetControlSurfaceSettings()
CFD_FM_Socket_Iface.Send( ctrlSurfSettings )
engineSettings = FM_Solver.GetEngineSettings()
CFD_FM_Socket_Iface.Send( engineSettings )

```

```
while True:
    # Empf. CFD-Lasten und uebertrage sie an FM_Solver
    loads = CFD_FM_Socket_Iface.Receive()
    FM_Solver.SetLoads(loads)

    # Uebertrage von FM_Solver nach Simulink,
    # Fuehre Simulink-Rechnung aus und
    # uebertrage Simulink-Loesung zurueck an FM_Solver
    FM_Solver.Run()

    # Sende Signal, ob Rechnung fortgesetzt oder abgebrochen
    # werden soll
    if FM_Solver.GetSignal() == FM_Solver.STOP_SIGNAL:
        CFD_FM_Socket_Iface.Send( FM_Solver.STOP_SIGNAL )
        break
    else:
        CFD_FM_Socket_Iface.Send( FM_Solver.CONTINUATION_SIGNAL )

    # Sende Bewegungszustand, Steuerflaechen- und Triebwerkseinstellung
    motion = FM_Solver.GetMotion()
    CFD_FM_Socket_Iface.Send( motion )
    ctrlSurfSettings = FM_Solver.GetControlSurfaceSettings()
    CFD_FM_Socket_Iface.Send( ctrlSurfSettings )
    engineSettings = FM_Solver.GetEngineSettings()
    CFD_FM_Socket_Iface.Send( engineSettings )
# Ende der Zeitschleife
```

Die Klasse *FM_Solver*

Innerhalb des Konstruktors von *FM_Solver* wird als erstes MATLAB/Simulink als Subprozess von Python aus gestartet. Dazu wird der Befehl aus Programmausdruck 2.8 verwendet. *FMTA_TAU_Trim* repräsentiert in diesem Aufruf das MATLAB-Skript (d. h. das M-File), das gestartet werden soll. Es wurde dem Konstruktor als Argument übergeben (s. Programmausdruck 2.6). Das angestartete MATLAB-Skript enthält den MATLAB/Simulink-Prozess des FMTA-Modells, der entweder eine Trimmrechnung oder eine flugdynamische Simulation realisiert.

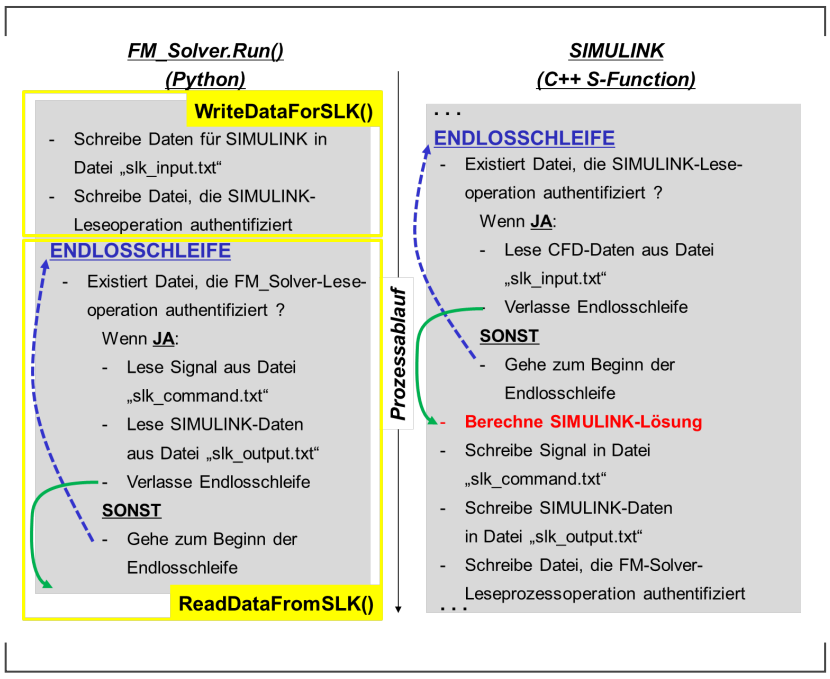
Programmausdruck 2.8: **Anstarten von MATLAB/Simulink aus Python**

```
import subprocess as sp

cmd = 'matlab -r "FMTA_Tau_trim"'
matlab = sp.Popen(cmd, shell=True, stdout=sp.PIPE, stderr = sp.PIPE)
```

FM_Solver besitzt alle Größen, die zwischen Simulink und der CFD-Seite ausgetauscht werden müssen als Membervariablen (Bewegungszustand, Steuerflaechen- und Triebwerkseinstellungen, aerodynamische Kräfte und Momente, Atmosphärenzustand, etc.). Für die

Abbildung 2.7: **Prinzipieller Ablauf der realisierten Datei-basierten Kopplung zwischen Python und Simulink.**

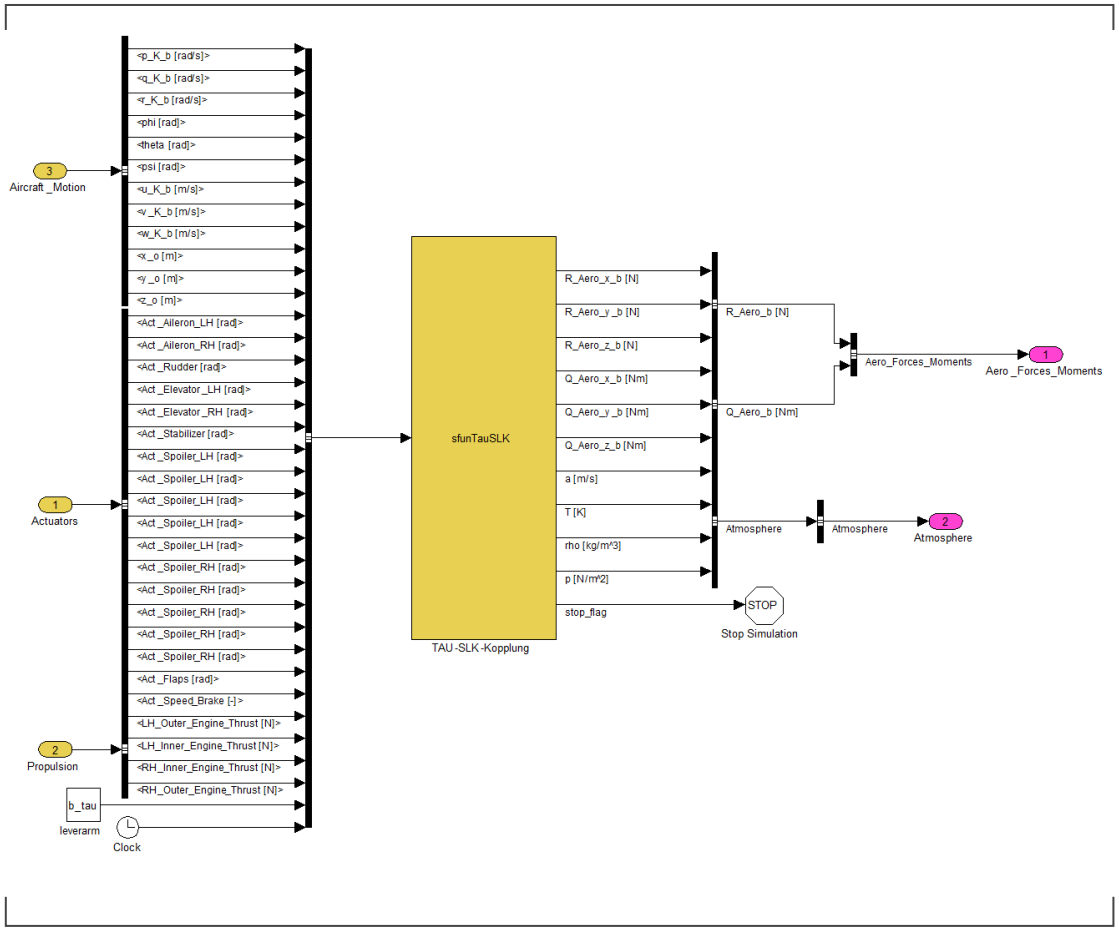


aerodynamischen Lasten und den Atmosphaerenzustand, d. h. für Daten, die an Simulink zu übertragen sind, bietet *FM_Solver* eine separate *Set*-Methode. Für Daten, die an die CFD-Seite gesendet werden müssen, sind *Get*-Methoden implementiert, um den Zugriff auf die Membervariablen zu realisieren.

Der eigentliche datei-basierte Datenaustausch zwischen *FM_Solver* und Simulink geschieht in den Methoden *ExchangeInitData* und *Run*, die in den Zeilen 7 und 29 des Programmausdrucks 2.7 aufgerufen werden. Beide Methoden sind annähernd identisch. Prinzipiell erfolgt der Datenaustausch entsprechend der Darstellung in Abb. 2.7. Die Methode *Run* ruft erst die private Methode *WriteDataForSLK*, die die Daten, die mit Simulink auszutauschen sind (siehe Tab. B.2), in eine Zeile der Datei *slk_input.txt* schreibt. Nach Abschluss dieser Operation schreibt *WriteDataForSLK* eine zweite Datei, die Simulink signalisiert, dass CFD-Daten gelesen werden können.

Simulink befand sich derweil in einer Endlosschleife, die diese infolge des Signals, dass Daten gelesen werden können, verlassen wird (siehe rechten Teil der Abb. 2.7). Danach führt Simulink einen Berechnungsschritt aus, schreibt die Daten, die mit der CFD-Seite ausgetauscht werden müssen (siehe Tab. B.1), in eine Zeile der Datei *slk_output.txt* und schreibt eine weitere Datei, die dem *FM_Solver* signalisiert, dass Daten gelesen werden

Abbildung 2.8: **Anbindung der SFunction an das FMTA-Simulink-Modell (Modellebene 3).**



können. Diese Daten werden danach von der privaten Methode *ReadDataFromSLK* des *FM_Solver* ausgelesen.

Modifikationen des FMTA-Simulink-Modells zur Kopplung mit TAU

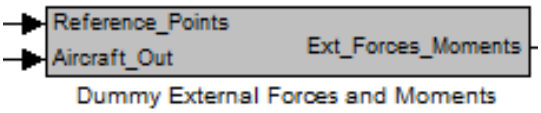
Das Simulink-Modell des FMTA wurde in folgender Weise für die Kopplung mit TAU erweitert. Zum einen musste der datei-basierte Datenaustausch mit Python (d. h. mit dem *FM_Solver*) realisiert werden. Andererseits musste die Verarbeitung der von TAU übertragenen aerodynamischen Kräfte und Momente ins Modell integriert werden.

Das Programm Simulink bietet innerhalb einer ununterbrochenen Simulation nicht die Funktionalität als vorgefertigten "Block", die zur Realisierung des datei-basierten Datenaustauschs entsprechend Abb. 2.7 benötigt wird. Daher wurde in das FMTA-Modell eine sogenannte C++-SFunction implementiert (siehe Abb. 2.8), die diese Funktionalität gewährleistet.

Der zentrale Block *TAU-SLK-Kopplung* (gelber "Kasten" in Abb. 3) beinhaltet die C++-SFunction *sfunTauSLK.cpp*. Für die korrekte Berechnung der aerodynamischen Kräfte und Momente benötigt die CFD-Seite die in Tab. B.1 (S. 59) aufgelisteten Größen. Diese werden von der rechten Seite (siehe Abb. 2.8) her von Simulink an den SFunction-Block übergeben. Für die flugdynamische Berechnung benötigt Simulink die in Tab. B.2 (S. 61) aufgelisteten Größen, welche von der linken Seite des SFunctions-Blockes an Simulink übermittelt werden.

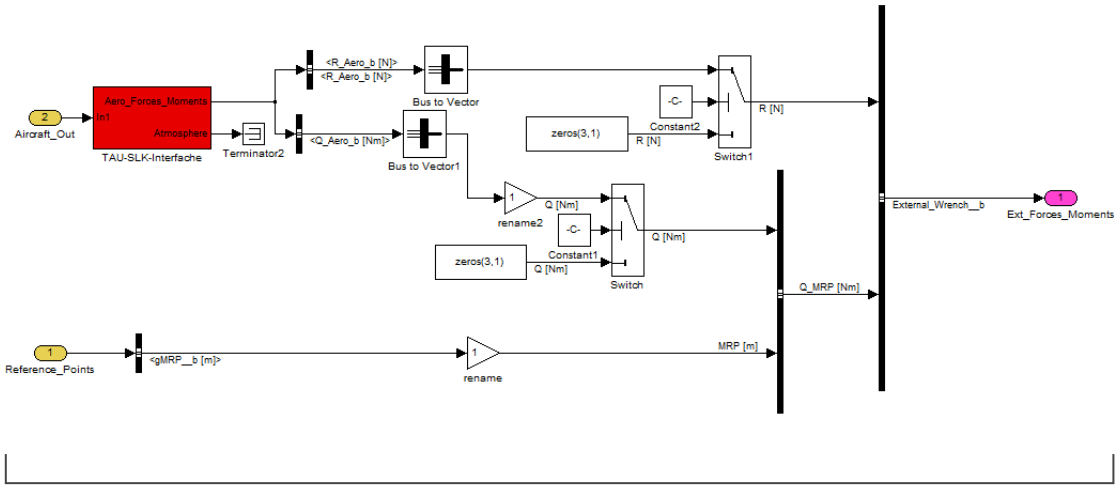
Das Simulink-Modell des FMTA in der Version V3 bietet für das Aufbringen von externen Kräften auf der obersten Modellebene (Modellebene 1) eine einfache Schnittstelle (siehe Abb. 2.9). Diese Schnittstelle erhält als Eingänge zum einen den Referenzpunkt, in den die Kräfte und Momente eingebracht werden, und zum anderen alle relevanten flugmechanischen Größen. Zusätzlich wurde in der darunter liegenden Modellebene ein Schalter zum Einschalten und Ausschalten der Kopplung mittels der Variablen *tau_simulation_flag* implementiert (siehe Abb. 2.10). Ist diese Variable Null so wird das ursprüngliche Derivativmodell des FMTA-Simulink-Modells verwendet und die externen Kräfte und Momente werden zu Null gesetzt. Wird die Variable *tau_simulation_flag* auf den Wert Eins gesetzt, so werden die externen Kräfte und Momente verwendet und diejenigen des Derivativmodells zu Null gesetzt.

Abbildung 2.9: **Schnittstelle im Simulink-Modell des FMTA für das Aufbringen von externen Kräften und Momenten (Modellebene 1).**



Da z. B. die Aktuatoren der Steuerflächen unter Simulink als PT2-Glieder modelliert wurden, ist es nicht möglich, Simulink nach jedem Aufruf durch das Python-Skript zu schließen und für die nächste Berechnung erneut aufzurufen. In diesem Fall würden nämlich z. B. die Zustände der Integratoren verloren gehen, was zu falschen Ergebnisse führen würde. Es ist also notwendig, das FMTA-Modell ohne das Beenden von Simulink *laufen* zu lassen. Die in dem SFunction-Block *TAU-SLK-Kopplung* enthaltene C++-SFunction realisiert das Warten der Simulation auf den Sendevorgang der CFD-Seite. Die detaillierte Logikstruktur der C++-SFunction ist in Abb. 2.11 dargestellt.

Abbildung 2.10: **Anbindung des TAU-SLK-Interfaces an den Block im Simulink-Modell des FMTA zum Aufbringen von externen Kräften und Momenten (Modellebene 2).**



Wenn das FMTA-Simulink-Modell im Modus *Trimmrechnung* läuft, wird bei jedem Iterationsschritt des Trimmalgorithmus ein Austausch mit der CFD-Seite vorgenommen. Im Modus *Flugdynamische Simulation* kommt es nur bei jedem physikalischem Zeitschritt zu einem Austausch der Daten.

Durch einige Eigenschaften der CFD-Seite, musste der Trimmalgorithmus im FMTA-Simulink-Modell wie folgt angepasst werden:

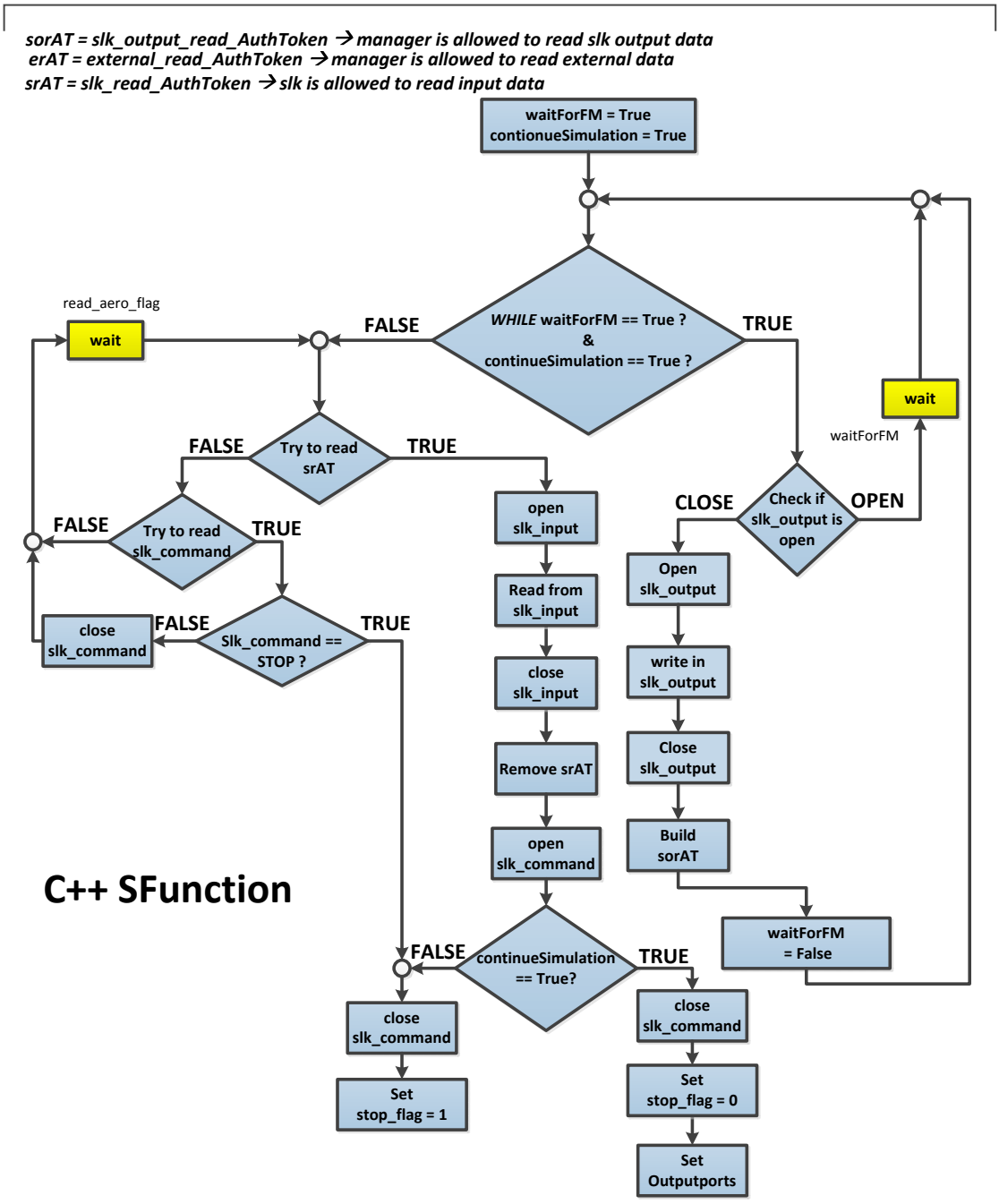
- Es mussten größere Variationen der Trimmgrößen zugelassen bzw. ausgeführt werden.

- Eine geringfügig schlechtere Kostenfunktion als bei Nutzung des Derivativmodells üblich musste akzeptiert werden.

- Der Trimmungalgorithmus wurde auf die Trimmung der Längsbewegung beschränkt.

Die Anpassungen haben allerdings keinen Einfluss auf die grundsätzliche Funktion der Kopplung.

Abbildung 2.11: **Detailliertes Logikplan der Implementierung der C++-SFunction *sfunTauSLK.cpp*.**



3 Anwendung der Kopplung auf FMTA-Konfiguration mit Beschädigung

3.1 Beschädigungsszenario und Simulationsablauf

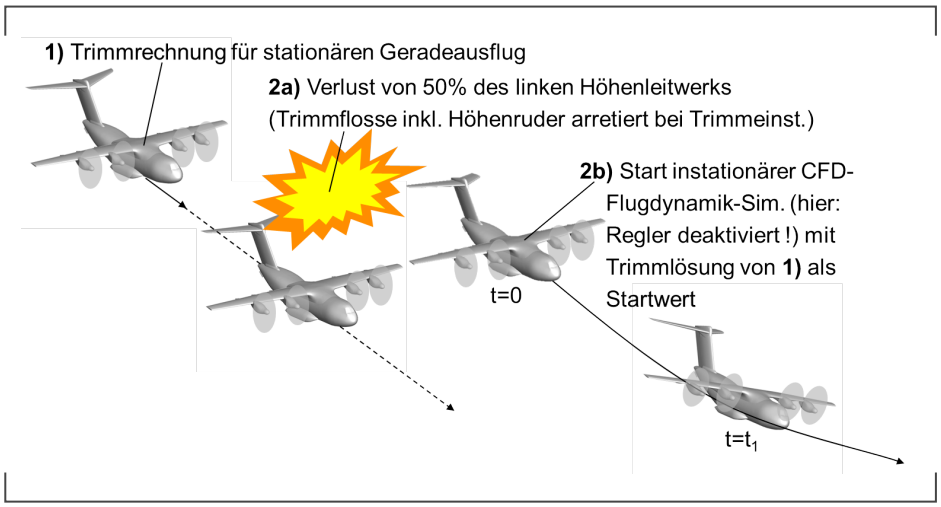
Als Beschädigungsszenario wurde der Verlust von 50% der linken Trimmflosse ausgewählt. Es wird angenommen, dass der Verlust plötzlich beim folgenden Flugzustand am ausgetrimmten FMTA auftritt: Fluggeschwindigkeit 270kn (IAS), Flugzeugmasse 120.17t, Flughöhe 4000ft.

Das gewählte Vorgehen zur Simulation des Beschädigungsszenarios ist in Abb. 3.1 illustriert. Es wird zunächst eine Trimmsimulation der nicht beschädigten Konfiguration unter Nutzung der entwickelten CFD-Simulink-Kopplung durchgeführt. Die anschließende Flugdynamiksimulation mit der beschädigten Konfiguration wird mit dem Trimmzustand der nicht degradierten Konfiguration initiiert. Aufgrund des dann bestehenden Kraft- und Momentenungleichgewichts führt die Konfiguration eine beschleunigte Bewegung durch, die mit der CFD-Simulink-Kopplung im Flugdynamiksimulationsmodus berechnet wird. Bei der Simulation wird bewusst das ungeregelte FMTA betrachtet.

3.2 CFD-Netze

Entsprechend dem genannten Vorgehen wurden Strömungsnetze für die nicht beschädigte und die beschädigte Konfiguration benötigt. Im AP wurden daher die in Abb. 3.2 gezeigten Euler-Netze mit niedriger Auflösung für beide Konfigurationen mit dem Netzgenerator CENTAUR erstellt. Beide Netze unterscheiden sich lediglich in dem in Abb. 3.3

Abbildung 3.1: **Simulationsablauf für ausgewähltes Beschädigungsszenario.**



dargestellten sehr begrenzten Bereich um die Beschädigungsstelle herum. Der Bereich entspricht der in Abb. 1.2 dargestellten blauen Box, die in CENTAUR zur modularen Netzgenerierung definiert wurde.

Die farbigen Netzanteile in Abb. 3.2 markieren die im Netz als beweglich definierten Steuerflächen. Spoiler sind in diesem Fall CFD-seitig ebenso wenig repräsentiert wie die Hochauftriebshilfen. In der Realität bestehende Spalte zwischen Steuerflächen und restlichen Auftriebsflächen sind ebenfalls nicht berücksichtigt (siehe Kap. 3.3 *Realisierung von Steuerflächenausschlägen in der CFD-Simulation*). Welche TAU-Marker im vorliegenden Fall mit welchen Oberflächennetzanteilen assoziiert sind, ist im Anhang A in Tab. A.1 (S. 56) angegeben.

Die aerodynamische Wirkung der Triebwerke ist CFD-seitig ebenfalls zur Reduktion des Berechnungsaufwands nicht berücksichtigt, weder durch direkte Auflösung der Propeller im Netz, noch in Form von Triebwerkswirkscheiben.

Der Funktionsnachweis der Kopplung bleibt von allen genannten Vereinfachungen jedoch unbeeinflusst. Auf Seiten der Flugdynamiksimulation kommandierte Änderungen der Triebwerkseinstellungen werden über die Schnittstelle kommuniziert (siehe Programmausdruck 2.4) und könnten ohne weiteres bei vorhandener Triebwerksrepräsentierung in der CFD-Simulation mit berücksichtigt werden. Auf Seiten des Simulink-Modells wird die Schubwirkung der Triebwerke in der Kraft-/Momentenbilanz allerdings stets voll berücksichtigt.

Abbildung 3.2: **Generierte grobe CFD-Netze (Euler-Netze) für ausgewähltes Beschädigungsszenario.**

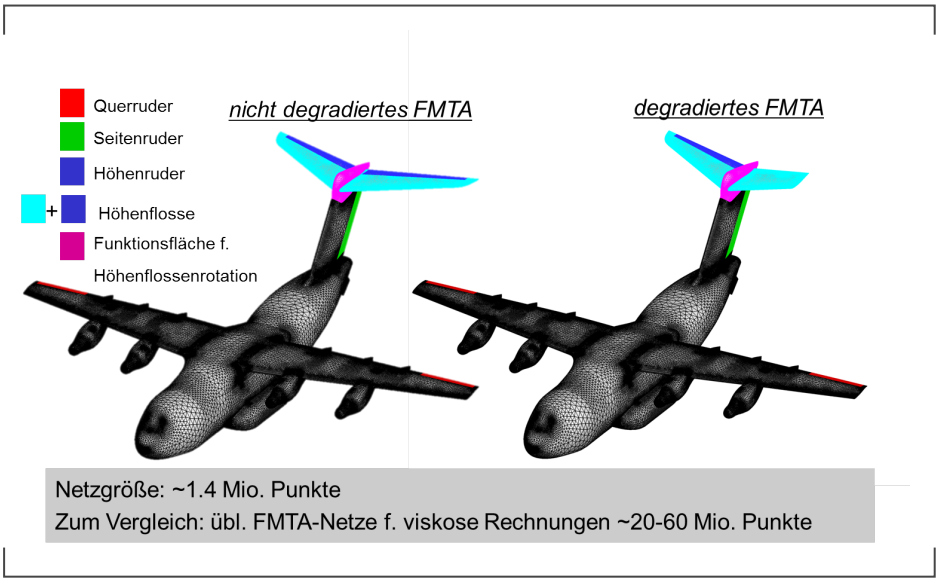


Abbildung 3.3: **Detailansicht auf die Netzunterschiede im Bereich der Beschädigung.**

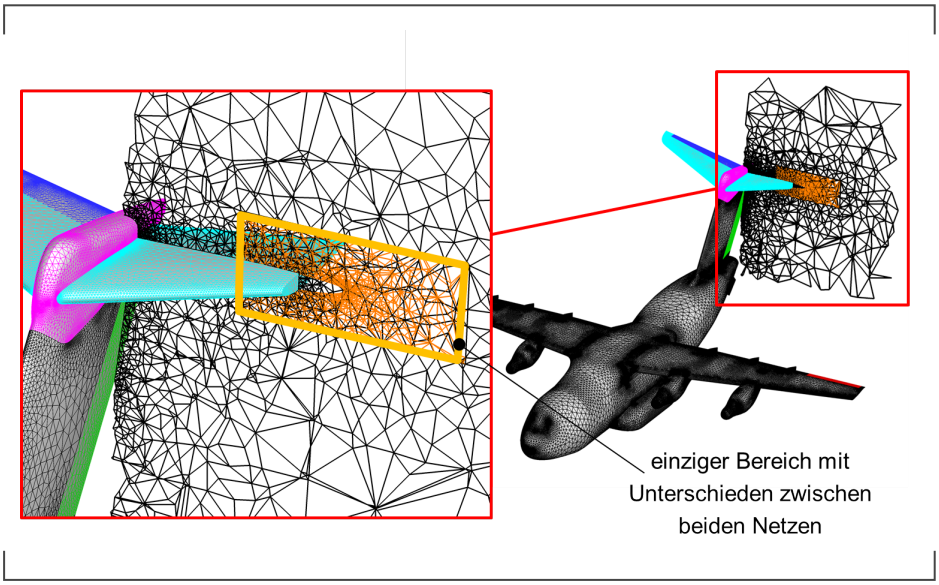
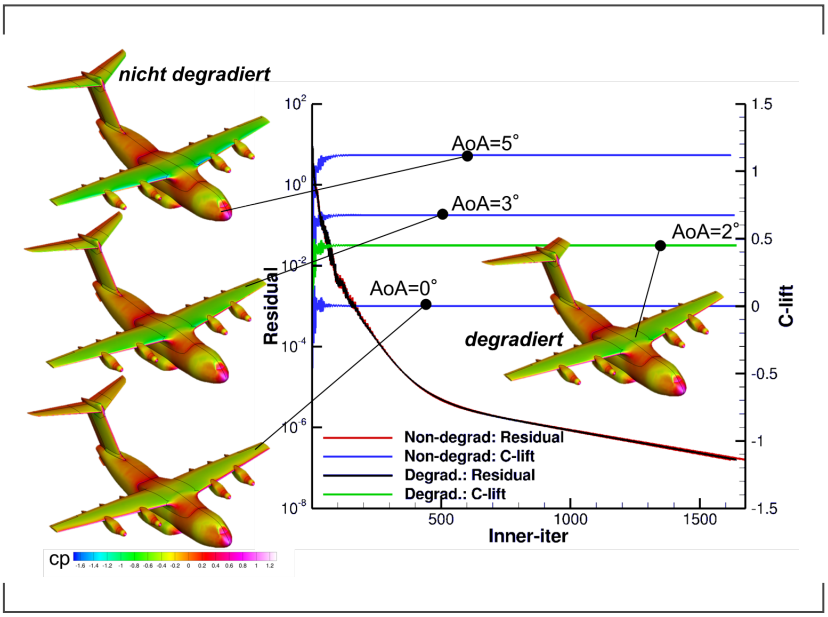


Abbildung 3.4: **Vergleich des TAU-Konvergenzverhaltens für das CFD-Netz der unbeschädigten Konfiguration mit demjenigen der beschädigten Konfiguration.**



Beide generierte Netztypen wurde im vorhinein zu den gekoppelten Simulationen daraufhin getestet, ob auf ihnen eine ausreichende Konvergenz des Strömungslösers zu erzielen ist. Für eine Reihe von Anstellwinkeln wurden daher stationäre Simulationen mit TAU mit den Einstellungen aus Kap. 3.4 durchgeführt. Abbildung 3.4 demonstriert, dass in allen untersuchten Fällen hinreichend gute Konvergenzeigenschaften des CFD-Lösers auf den gegebenen Netzen zu erzielen ist.

3.3 Realisierung von Steuerflächenausschlägen in der CFD-Simulation

Aus Effizienzgründen wurden im Rahmen des Projekts Steuerflächenausschläge CFD-seitig mittels Deformation des CFD-Netzes realisiert. Andersartige Realisierungen (z. B. mittels der Chimera-Technologie) sind ebenfalls möglich, wurden aber aus Gründen der Aufwandsreduktion im Projekt nicht verfolgt. Die entwickelte CFD-Simulink-Kopplung bleibt von dieser Beschränkung allerdings unbeeinflusst. Sie ist auch bei andersartigen CFD-seitigen Realisierungen von Steuerflächenausschlägen uneingeschränkt einsetzbar.

Nachfolgend wird erläutert, in welcher Weise unter Nutzung von *FlowSimulator*-Funktionalitäten kommandierte kombinierte Ausschläge von Trimmflosse und primären Steuerflächen (Querruder, Höhenruder, Seitenruder) in der Simulation realisiert wurden.

Für einen kombinierten Ausschlag müssen zunächst die Koordinaten des unverformten CFD-Ausgangsnetz gespeichert werden. Die Realisierung dessen zeigt Programmausdruck 3.1. Die Implementierung des *UnstructDatasetHandler* (Z. 1 aus Programmausdruck 3.1) ist im Anhang D.3 im Programmausdruck D.5 (S. 77) angegeben.

Programmausdruck 3.1: **Speicherung der Koordinaten des unverformten CFD-Netzes.**

```
1 DatasetHandler = UnstructDatasetHandler(CFDMesh)
2 xyzJig = DatasetHandler.Get(FSDataName.Coordinates())
```

3.3.1 Verstellung der Trimmfläche

Zur Verstellung der Trimmflosse (cyan- und blau-gefärbte Oberflächennetzanteile in Abb. 3.2) ist dem Netzdeformationsverfahren *FSDeformation* ein geeignetes Verschiebungsfeld vorzugeben, dass der gewünschten Starrkörperdrehung der Trimmflosse entspricht. Zur bequemen Berechnung des Verschiebungsfeldes — alleinig auf der Angabe von Rotationsachse und dem Rotationswinkel — wurde die Klasse *SurfMeshRotationHandler* implementiert. Ihre genaue Implementierung ist im Anhang D im Progammausdruck D.4 (S. 75) angegeben. Die Nutzung der Klasse zeigt Programmausdruck 3.2.

Programmausdruck 3.2: **Instanzierung eines Objekts zur Generierung eines Verschiebungsfeldes für eine Starrkörperrotation mehrerer Oberflächennetzsegmente.**

```
1 HTPHandler = SurfMeshRotationHandler(DM)
2 htpSurfMesh = HTPHandler.GetAndRegisterSurfaceMesh(MeshKey="CFDMesh",
                                                     SurfMeshKey="HTPSurfMesh",
4                                                    CADGroupIDs=markersHtp)
```

Programmausdruck 3.2 zeigt, wie aus dem CFD-Volumennetz, das durch den Namen "CFDMesh" im *FSDataManager* ansprechbar ist, das Oberflächennetz extrahiert wird, das die TAU-Marker aus der Python-Liste *markersHtp* umfasst, und wie es unter dem Namen "HTPSurfMesh" im *FSDataManager* registriert wird.

Anschließend wird eine Instanz der CFD-Volumennetzdeformationsmethode *FSDeformation* mit geeigneten Randbedingungen für die Trimmflossenerstellung erzeugt. Dies zeigt

Programmausdruck 3.3. Darin werden zunächst die TAU-Marker der Oberflächennetzanteile bestimmt, die im Rahmen der Netzdeformation festgehalten werden. Dies sind alle TAU-Marker exklusive denen von Trimmflossenfunktionsfläche und Trimmflosse selbst. Es werden drei Randbedingungen erstellt. Die erste führt dazu, dass die betroffenen Oberflächennetzanteile fixiert bleiben. Die zweite legt eine spezielle Randbedingung (*BT_NoNormalMovement*) für die Funktionsfläche fest. Sie gewährleistet, dass das Netz auf der Funktionsfläche des T-Leitwerks des FMTA entlang der *diskretisierten* Funktionsfläche *gleitet*. Die Leitwerksgeometrie bleibt bei Nutzung dieser Randbedingung annähernd gewahrt, sofern die Ausschläge innerhalb bestimmter Ausschlagsgrenzen bleiben. Die dritte Randbedingung bezieht sich auf die Trimmflosse selbst.

Programmausdruck 3.3: **Instanzierung eines Netzdeformationsobjektes für die Verstellung der Trimmfläche.**

```
#Bestimmung der Marker fuer die festgehaltenen Oberflaechensegmente
markersFixed4HtpRot      = markersWholeAC[:]
for mDel in markersHtp + markersHtpRotationSurf:
    markersFixed4HtpRot = filter(lambda mInList: mInList != mDel, markersFixed4HtpRot)

# Verschiebungsfeld (4 Raumpunkte mit Null-Verschiebungen) fixiert betreffendes
    Bauteils
fixedpoints = np.zeros((4,3), np.float64)
fixedpoints[1,0] = 1.
fixedpoints[2,1] = 1.
fixedpoints[3,2] = 1.
zerodispl   = np.zeros((4,3), np.float64)

FSMeshDefo4HTPRot = FSDeformation(CFDMesh, para=generalMeshDefoParas)

# Randbedingung fuer fixierte Oberflaechenanteile
FSMeshDefo4HTPRot.AddGroup({"Fixed4HTPRot":
                      {"TargetBoundaryValues"           : markersFixed4HtpRot,
                       "BoundaryType"                   : BT_Standard,
                       "MaxBasePoints"                  : 4,
                       "RadialBasisFunction"            : "cubic-volume-spline",
                       "ReductionMethod"                : RM_Equidistant,
                       "RadiusFullWeight"               : 0.8,
                       "RadiusZeroWeight"               : 20.,
                       "UseFSDMDeformationVectorInput"  : 1,
                       "FSDMDeformationVectorBasePoints": fixedpoints,
                       "FSDMDeformationVectors"         : zerodispl}
                      })

# Randbedingung fuer Funktionsflaeche
FSMeshDefo4HTPRot.AddGroup({"NNM":
                      {"TargetBoundaryValues"           : markersHtpRotationSurf,
                       "BoundaryType"                   : BT_NoNormalMovement,
                       "MaxBasePoints"                  : 2000,
                       "RadialBasisFunction"            : "cubic-volume-spline",
                       "ReductionMethod"                : RM_ErrorWeighting,
                       "RadiusFullWeight"               : 0.8,
                       "RadiusZeroWeight"               : 20.,
```

```
                        "NNM_Steps"                         : 100,
                        "NNM_ChangeOverLength"              : 0.2}
                       })

# Randbedingung duer Trimmflosse
FSMeshDefo4HTPRot.AddGroup({"HTP":
                       {"TargetBoundaryValues": markersHtp,
                        "BoundaryType": BT_Standard,
                        "MaxBasePoints": 1000,
                        "RadialBasisFunction": "cubic-volume-spline",
                        "ReductionMethod": RM_Equidistant,
                        "RadiusFullWeight" : 0.8,
                        "RadiusZeroWeight" : 20.0,
                        "NNM_RadiusFullWeight":0.1,
                        "NNM_RadiusZeroWeight":0.5,
                        "UseFSDMDeformationVectorInput": 1}
                       })
```

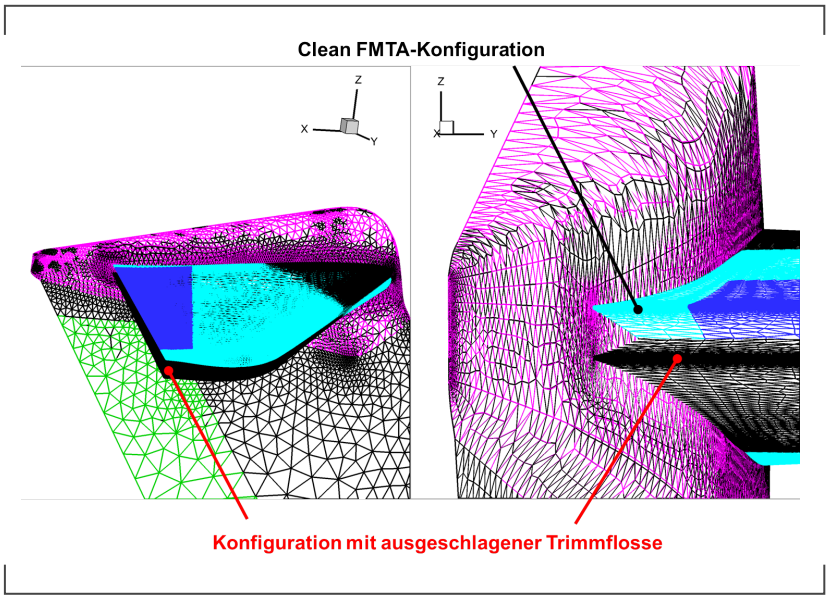
Für die Trimmflosse ist unter Nutzung des Objekts aus Progammausdruck 3.2 das jeweils zur Trimmflossenrotation passende Verschiebungsfeld zu generieren. Wie dies realisiert ist, zeigt Programmausdruck 3.4. *htpDeflAngle* (Z. 5) ist darin die von der Simulink-Prozessseite kommandierte Trimmflossenrotation. *HingePoint* und *Axis* definieren die Lage der Rotationsachse. Der *HTPHandler* gibt die Basispunkte und Verschiebungsvektoren zurück die der vorgegeben Rotation entsprechen. Diese werden dann in der Netzdefomrationsrandbedingung für die Trimmflosse gesetzt. Anschließend wird die eigentliche Volumennetzdeformation durchgeführt (Z. 12 in Programmausdruck 3.4). Zum Schluss werden die Verschiebungsvektoren aller CFD-Volumennetzpunkte gespeichert, die sich infolge der Trimmflossenverstellung ergeben haben (Z. 15).

Programmausdruck 3.4: **Berechnung des Verschiebungsfeldes aller CFD-Volumennetzpunkte bei vorgegebener Verstellung der Trimmfläche.**

```
1  htpBaseCoordsNumpy, htpDisplVecsNumpy = \
2      HTPHandler.GetDefoVectorsForRotation(SurfMeshKey="HTPSurfMesh",
                                            HingePoint=[38.854263,0.,10.382694],
4                                           Axis=[0,1,0],
                                            Angle=htpDeflAngle)
6  FSMeshDefo4HTPRot.ModifyGroup(
       {"HTP":
8           {"FSDMDeformationVectorBasePoints": htpBaseCoordsNumpy,
            "FSDMDeformationVectors"          : htpDisplVecsNumpy}
10      })

12 FSMeshDefo4HTPRot.ChainRun(Clac)
13
14 xyzWithDeflHTP = DatasetHandler.Get(FSDataName.Coordinates())
   dxyzWithDeflHTP = xyzWithDeflHTP - xyzJig
```

Abbildung 3.5: **Ausschlag der Trimmflosse um +4° mittels Netzdeformation (Anwendung der Gleitrandbedingung *BT_NoNormalMovement*) in *FSDeformation*. Das schwarze Netz ist das, in dem die Trimmflosse ausgeschlagen ist.**



In Tests konnte die Trimmflosse bei den generierten CFD-Netzen im Bereich zwischen +/-4.5° ausgeschlagen werden, ohne dass Zellen mit negativem Volumen im Netz auftraten, die ansonsten einen Abbruch der CFD-Simulation zur Folge gehabt hätten. Abb. 3.5 zeigt exemplarisch das Oberflächennetz auf dem Höhenleitwerk für einen mit Netzdeformation erzielten Trimmflossenausschlag von +4°.

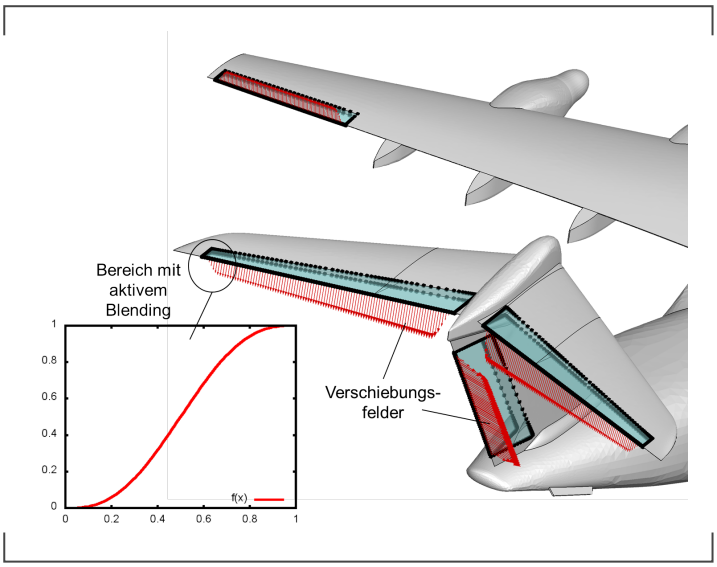
Für Beschreibungen zu *FSDeformation*-Parametern, die in den Programmausdrücken 3.3 und 3.4 verwendet werden, wird auf das Handbuch von *FSDeformation* verwiesen [6].

3.3.2 Verstellung primärer Steuerflächen

Im Rahmen des Projekts wurde zusätzlich eine effiziente Methode geschaffen (*FSCtrlSurfDisplFieldGenerator*[1]), um die restlichen Steuerflächen (ausgenommen die Spoiler) mittels Netzdeformation auszuschlagen. Basierend auf den Angaben über die Drehachse und den

[1] Zugang zum Quellcode unter `https://dev2.as.dlr.de/svn/fsdeformation/trunk/FSCtrlSurfDisplFieldGenerator`

Abbildung 3.6: **Verschiebungsfelder generiert von *FSCtrlSurfDisplFieldGenerator* als Eingangsdaten für die Verstellung der Steuerflächen mittels Netzdeformation (*FSDeformation*). Die Verschiebungsfelder werden basierend auf dem angegebenen Ausschlagswinkel und der Drehachse berechnet.**



Ausschlagwinkel werden automatisch entsprechende Verschiebungsfelder von der Methode generiert, die im Rahmen der Volumennetzdeformation *FSDeformation* zur Verstellung der jeweiligen Steuerfläche verwendet werden können. Die Methode ist anwendbar, wenn keine Steuerflächenspalte CFD-seitig berücksichtigt werden. Derartige Steuerflächenrepräsentierungen liefern allerdings meist bereits eine ausreichend gute Simulation der integralen Wirkung des jeweiligen Steuerorgans. Zur Vermeidung arg verschlechterter Netzqualität im Übergangsbereich zwischen ausgeschlagener Steuerfläche und restlicher Auftriebsfläche wird in dem betreffenden Bereich automatisch ein sogenanntes *Blending* durchgeführt, so dass ein kontinuierlicher Übergang zwischen Klappennetz und restlichem Oberflächennetz entsteht.

Abbildung 3.6 zeigt exemplarisch die von *FSCtrlSurfDisplFieldGenerator* generierten Verschiebungsfelder für kombinierte Ausschläge von Querruder, Seitenruder und Höhenruder. Die Verschiebungsfelder enthalten Basispunkte mit Nullverschiebungen an den Rändern des jeweiligen Oberflächensegments — sie gewährleisten einen glatten Übergang — und von Null verschiedene Verschiebungen entlang der approximierten Hinterkante der Steuerfläche. In der Abbildung ist ebenfalls die an den seitlichen Rändern der jeweiligen Steuerfläche verwendete Blendingfunktion dargestellt, die entlang der Hinterkante zum Steuerflächenrand hin vom Wert 1 auf den Wert 0 übergeht.

Die Verschiebungsfelder aus Abb. 3.6 wurden wie folgt mit *FSCtrlSurfDisplFieldGenerator* berechnet. Zunächst muss für jede Steuerfläche, die verstellt werden soll, eine Instanz von *FSCtrlSurfDisplFieldGenerator* gebildet werden. Für den konkreten Fall des FMTA zeigt dies der Programmausdruck 3.5. Für jede Steuerfläche sind im wesentlichen die TAU-Marker anzugegeben, die das Oberflächennetz der Steuerfläche bilden und zwei Punkte, die die Scharnierachse definieren. Welcher Marker welches Oberflächennetzsegment identifiziert, ist für die hier verwendeten CFD-Netze in Tab. A.1 (S. 56) angegeben. Die Lage der Scharnierpunkte ist in Tab. A.2 (S. 57) aufgelistet und in Abb. A.1 (S. 56) in Bezug zum FMTA anschaulich illustriert.

Zur Erläuterung der Wirkung der weiteren Parameter, die im Programmausdruck 3.5 bei jeder Instanzierung von *FSCtrlSurfDisplFieldGenerator* angegeben sind, wird auf das Handbuch von *FSCtrlSurfDisplFieldGenerator* verwiesen [7].

Programmausdruck 3.5: **Instanzierung von "Verschiebungsfeldgeneratoren" für jede Steuerfläche.**

```
#### RIGHT AILERON ####
AileronRightHandler = CtrlSurfDisplFieldGenerator(\
    DM,
    "CFDMesh",
    markersAileronR,
    [20.88471,16.03119,2.666916],
    [21.32592,20.88471,2.45868],
    1000,
    0.1,
    ctrlSurfName=ailRightTV.GetName())

basePointsAileronR = AileronRightHandler.GetBasePointsAsNumpy()

#### LEFT AILERON ####
AileronLeftHandler = CtrlSurfDisplFieldGenerator(\
    DM,
    "CFDMesh",
    markersAileronL,
    [21.32592,-20.88471,2.45868],
    [20.88471,-16.03119,2.666916],
    1000,
    0.1,
    ctrlSurfName=ailLeftTV.GetName())

basePointsAileronL = AileronLeftHandler.GetBasePointsAsNumpy()

#### VERTICAL TAIL PLANE ####
RudderHandler = CtrlSurfDisplFieldGenerator(\
    DM,
    "CFDMesh",
    markersRudder,
    [40.477101,0.,9.419382],
    [36.56247,0.,3.975363],
    1000,
```

```
    0.1,
    ctrlSurfName=rudTV.GetName())

basePointsRudder = RudderHandler.GetBasePointsAsNumpy()

#### RIGHT ELEVATOR ####
ElevatorRightHandler = CtrlSurfDisplFieldGenerator(\
    DM,
    "CFDMesh",
    markersElevatorR,
    [40.18119,0.4683,10.327695],
    [43.963374,8.594439,10.358628],
    1000,
    0.08,
    ctrlSurfName=eleRightTV.GetName())

basePointsElevatorR = ElevatorRightHandler.GetBasePointsAsNumpy()

#### LEFT ELEVATOR ####
ElevatorLeftHandler = CtrlSurfDisplFieldGenerator(
    DM,
    "CFDMesh",
    markersElevatorL,
    [43.963374,-8.594439,10.358628],
    [40.18119,-0.4683,10.327695],
    1000,
    0.08,
    ctrlSurfName=eleLeftTV.GetName())

basePointsElevatorL = ElevatorLeftHandler.GetBasePointsAsNumpy()
```

Anschließend muss eine separate Instanz von *FSDeformation* erzeugt werden mittels der die Verschiebung der CFD-Volumennetzpunkte alleinig infolge der Steuerflächenausschläge berechnet wird. Jede auszuschlagenede Steuerfläche wird mit einer Standard-Netzdeformationsrandbedingung (Kennung *BT_Standard*) versehen, für die später Verschiebungsfelder vorzugeben sind.

Programmausdruck 3.6: **Instanzierung eines Netzdeformationsobjektes für die Verstellung der primären Steuerflächen.**

```
markersFixed4CSMotion    = markersWholeAC[:]
for mDel in markersAileronR + markersAileronL + markersRudder + markersElevatorL +
    markersElevatorR:
    markersFixed4CSMotion = filter(lambda mInList: mInList != mDel,
    markersFixed4CSMotion)

generalMeshDefoParas = {"DeformOnlySurfacePoints": 0,
                        "WriteAnalysisFiles": 0,
                        "PrintLevel": 5,
                        "UseBlendingOptimizedInterpolationMethod":1}

FSMeshDefo4CSMotion = FSDeformation(CFDMesh, para=generalMeshDefoParas)
```

```
#Group 1#########################
# AileronR
FSMeshDefo4CSMotion.AddGroup({AileronRightHandler.GetNameOfControlSurf():
                        {"TargetBoundaryValues"           : markersAileronR,
                         "BoundaryType"                   : BT_Standard,
                         "MaxBasePoints"                  : 1000,
                         "RadialBasisFunction"            : "cubic-volume-spline",
                         "ReductionMethod"                : RM_Equidistant,
                         "UseFSDMDeformationVectorInput"  : 1,
                         "RadiusFullWeight"               : 0.2,
                         "RadiusZeroWeight"               : 1.2}
                        })

#Group 2#########################
# AileronL
FSMeshDefo4CSMotion.AddGroup({AileronLeftHandler.GetNameOfControlSurf():
                        {"TargetBoundaryValues"           : markersAileronL,
                         "BoundaryType"                   : BT_Standard,
                         "MaxBasePoints"                  : 1000,
                         "RadialBasisFunction"            : "cubic-volume-spline",
                         "ReductionMethod"                : RM_Equidistant,
                         "UseFSDMDeformationVectorInput"  : 1,
                         "RadiusFullWeight"               : 0.2,
                         "RadiusZeroWeight"               : 1.2}
                        })

#Group 3#########################
# Rudder
FSMeshDefo4CSMotion.AddGroup({RudderHandler.GetNameOfControlSurf():
                        {"TargetBoundaryValues"           : markersRudder,
                         "BoundaryType"                   : BT_Standard,
                         "UseFSDMDeformationVectorInput"  : 1,
                         "MaxBasePoints"                  : 1000,
                         "RadialBasisFunction"            : "cubic-volume-spline",
                         "ReductionMethod"                : RM_Equidistant,
                         "RadiusFullWeight"               : 0.2,
                         "RadiusZeroWeight"               : 1.2}
                        })

#Group 4#########################
# ElevatorL (falls vorhanden !)
FSMeshDefo4CSMotion.AddGroup({ElevatorLeftHandler.GetNameOfControlSurf():
                       {"TargetBoundaryValues"           : markersElevatorL,
                        "BoundaryType"                   : BT_Standard,
                        "UseFSDMDeformationVectorInput"  : 1,
                        "MaxBasePoints"                  : 1000,
                        "RadialBasisFunction"            : "cubic-volume-spline",
                        "ReductionMethod"                : RM_Equidistant,
                        "RadiusFullWeight"               : 0.2,
                        "RadiusZeroWeight"               : 1.2}
                       })

#Group 5#########################
# ElevatorR
FSMeshDefo4CSMotion.AddGroup({ElevatorRightHandler.GetNameOfControlSurf():
```

```
                    {"TargetBoundaryValues"           : markersElevatorR,
                     "BoundaryType"                   : BT_Standard,
                     "UseFSDMDeformationVectorInput"  : 1,
                     "MaxBasePoints"                  : 1000,
                     "RadialBasisFunction"            : "cubic-volume-spline",
                     "ReductionMethod"                : RM_Equidistant,
                     "RadiusFullWeight"               : 0.2,
                     "RadiusZeroWeight"               : 1.2}
                    })

#Group 6#########################
# Rest
FSMeshDefo4CSMotion.AddGroup({"Fixed4CSMotion":
                    {"TargetBoundaryValues"           : markersFixed4CSMotion,
                     "BoundaryType"                   : BT_Standard,
                     "MaxBasePoints"                  : 4,
                     "RadialBasisFunction"            : "cubic-volume-spline",
                     "ReductionMethod"                : RM_Equidistant,
                     "RadiusFullWeight"               : 0.2,
                     "RadiusZeroWeight"               : 1.2,
                     "UseFSDMDeformationVectorInput"  : 1,
                     "FSDMDeformationVectorBasePoints": fixedpoints,
                     "FSDMDeformationVectors"         : zerodispl}
                    })
```

Die Berechnung der Verschiebungsfelder zu den von Simulink kommandierten Steuerflächenausschlägen mittels den zuvor definierten *FSCtrlSurfDisplFieldGenerator*s (Programmausdruck 3.5) zeigt Programmausdruck 3.7 (Zeilen 1-15). Die eigentliche Netzdeformationsberechnung erfolgt in Zeile 17. Die Speicherung des Verschiebungsfeldes geschieht in Zeile 20.

Programmausdruck 3.7: **Berechnung des Verschiebungsfeldes aller CFD-Volumennetzpunkte bei vorgegebener Verstellung der primären Steuerflächen.**

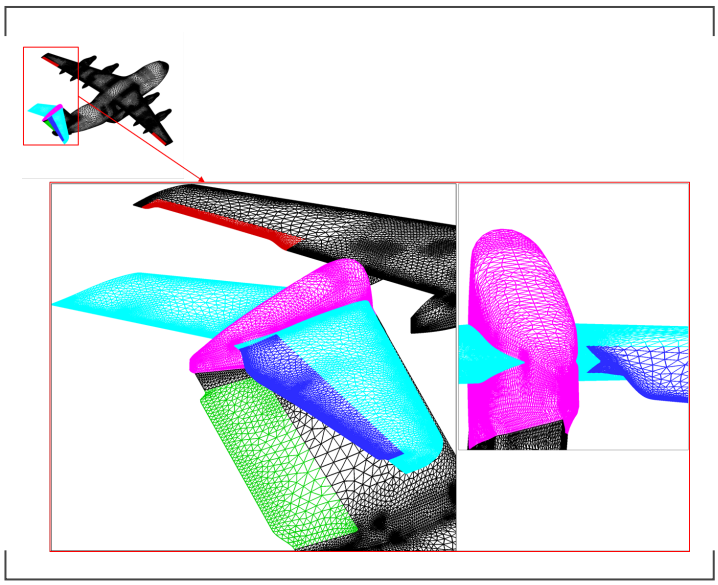
```
CtrlSurfDeflectionStates={AileronRightHandler: aileronRightDeflAngle,
                          AileronLeftHandler : aileronLeftDeflAngle,
                          ElevatorRightHandler: elevatorRightDeflAngle,
                          RudderHandler: rudderDeflAngle}

for handler, state in CtrlSurfDeflectionStates.items():
    deflAngle = float(state)

    baseCoordsNumpy, displVecsNumpy = \
        handler.GetBasePointsAndDisplacementVectorsForDeflectionAngleAsNumpy( deflAngle
     )
    FSMeshDefo4CSMotion.ModifyGroup(
        {handler.GetNameOfControlSurf():
             {"FSDMDeformationVectorBasePoints": baseCoordsNumpy,
              "FSDMDeformationVectors"         : displVecsNumpy}
         })

```

Abbildung 3.7: **Test kombinierter Ausschläge von Trimmflosse und restlichen Steuerflächen mittels Netzdeformation am beschädigten FMTA.**



```
FSMeshDefo4CSMotion.ChainRun(Clac)

xyzWithDeflCS = DatasetHandler.Get(FSDataName.Coordinates())
dxyzWithDeflCS = xyzWithDeflCS - xyzJig
```

Die Verschiebungsfelder der CFD-Volumennetzpunkte für die Trimmflossen- und die Steuerflächenverstellung werden final superponiert. Die Überlagerung wird zunächst, wie im Programmausdruck 3.8 (Z. 1) gezeigt, berechnet und danach in der Instanz des CFD-Netzes gesetzt (Z. 2).

Programmausdruck 3.8: **Kombination der Verschiebungsfelder für die Verstellung von Trimmfläche und primären Steuerflächen zu einem resultierenden Verschiebungsfeld der CFD-Volumennetzpunkte.**

```
1 xyzDefTotal = xyzJig + dxyzWithDeflCS + dxyzWithDeflHTP
2 DatasetHandler.Set(FSDataName.Coordinates(), xyzDefTotal)
```

Exemplarisch zeigt Abb. 3.7 die beschädigte FMTA-Konfiguration mit kombinierten Ausschlägen von Trimmflosse, Querruder, Höhenruder und Seitenruder, die mit den zuvor beschriebenen Techniken erwirkt wurden. Die Trimmflosse wurde um +4° ausgeschlagen, die primären Steuerflächen jeweils um 20°.

3.4 Trimmrechnung für nicht beschädigte Konfiguration

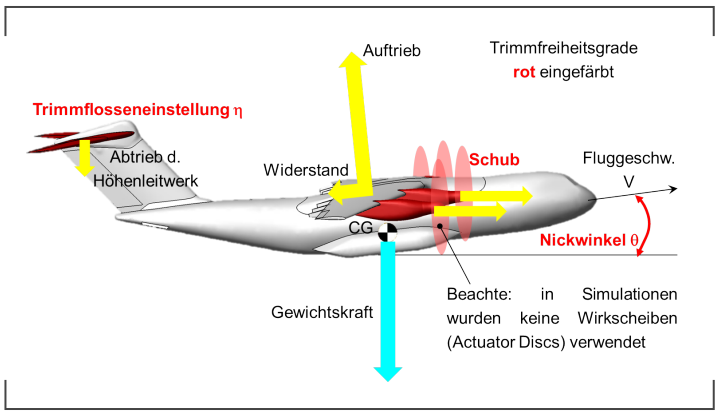
Für den genannten Flugzustand (siehe Kap. 3.1, S. 35) wurde mit der CFD-Simulink-Kopplung für die unbeschädigte FMTA-Konfiguration eine Trimmrechnung durchgeführt. Diese dient als initiale Simulation für die anschließende flugdynamische Simulation der beschädigten Konfiguration (siehe Erläuterungen in Kap. 3.1).

TAU (Version 2014.1.0) wurde mit folgenden Einstellungen im Rahmen der Trimmrechnung verwendet. Es wurde die zellecken-zentrierte Metrik angewendet. Zur Diskretisierung der konvektiven Flüsse wurde das zentrale Schema mit künstlicher Matrix-Dissipation genutzt. Das implizite Zeitschrittiterationsverfahren mit Euler-Rückwärtsdifferenz wurde zur Berechnung der stationären Lösung verwendet. Das entstehende Gleichungssystem für das linearisierte Problem wird mit einem LU-SGS-Verfahren *gelöst* (jeweils nur 1 Lösungsschritt pro *äußerer* nichtlinearer Iteration). Das Lösungsverfahren ist eingebettet in einen geometrischen Mehrgitteralgorithmus. Es wurde ein 3w-Mehrgitterzyklus mit einem Relaxationsschritt pro Mehrgitterebene eingesetzt. Im Sinne eines vollen Mehrgitterschemas wurde der Zyklus auf dem gröbsten Gitterniveau begonnen und maximal 300 Mehrgitterschritte vor dem Wechsel auf die nächst feinere Mehrgitterebene durchgeführt oder so viele Schritte ausgeführt bis das absoluten Residuum von 10^{-5} unterschritten wurde. Auf der feinsten Gitterebene und auf den Grobgitterebenen wurde über die gesamte Simulationszeit konstant CFL=5 verwendet.

Alle Simulationen wurden mit extern vorgegebenem Bewegungszustand und Anströmungsgeschwindigkeit Null am Fernfeldrand durchgeführt, d. h. das Flugzeug wurde — anders als sonst üblich — durch das ruhende Strömungsmedium bewegt.

Bei der Trimmrechnung wurde lediglich die Längsbewegung getrimmt. Dementsprechend wurden als Trimmfreiheitsgrade die Trimmflosseneinstellung η, der Triebwerksschub und der Nickwinkel θ betrachtet (siehe Abb. 3.8). Zusätzlich wurde im Trimmalgorithmus die Größe der Änderungen von Trimmvariablen von Iterationsschritt zu Iterationsschritt vergrößert, um die Anzahl der Iterationen zu minimieren. Diese Änderung war nötig, da kleine Änderungen u. U. keine Änderungen in der Berechnung der aerodynamischen Kräfte und Momente bewirkten. Die Änderung der residuellen Gesamtkräfte und -momente (links) sowie die Änderung der Trimmfreiheitsgrade (rechts) sind in Abb. 3.9 über die Anzahl der CFD-Simulink-Kopplungsschritte dargestellt. Entsprechend der Zielsetzung einer Trimmrechnung laufen die Kräftesummen F_x, F_z und die Nickmomentensumme M_y asymptotisch gegen Null. Die im Trimmpunkt erreichten Werte für den Nickwinkel und die Trimmflossenstellung sind in Abb. 3.9 (rechts) eingetragen. Der Nachweis der Funkti-

Abbildung 3.8: **Verwendete Freiheitsgrade zur Trimmrechnung der Längsbewegung.**



onsfähigkeit des Trimmsimulationsmodus der entwickelten CFD-Simulink-Kopplung wurde damit erfolgreich erbracht.

3.5 Aerodynamisch-flugdynamische Simulation der beschädigten Konfiguration

Wie im Kap. 3.1 (Beschädigungsszenario und Simulationsablauf) beschrieben, wurde die flugdynamische Simulation der beschädigten Konfiguration mit der Trimmlösung der unbeschädigten Konfiguration initiiert. Das Kräfte- und Momentenungleichgewicht, das in der Trimmbilanz infolge der Beschädigung entsteht, zieht erwartungsgemäß die nachfolgend genannten und in Abb. 3.10 illustrierten Bewegungen des FMTA nach sich. Aufgrund des einseitig reduzierten Abtriebs am Höhenleitwerk sollte das beschädigte FMTA in eine positive Rollbewegung übergehen. Der einseitig geringere Widerstand am Höhenleitwerk sollte eine positive Gierbewegung zur Folge haben. Ferner wird durch das geringere kompensatorische Nickmoment, das durch den teilweisen Verlust des Höhenleitwerks entsteht, eine kopflastige Nickbewegung erwartet.

In Abb. 3.11 (links) werden exemplarisch die zeitlichen Historien der vorhergesagten Lagewinkel und der vorhergesagte Höhenverlust für die unbeschädigte und die beschädigte FMTA-Konfiguration gezeigt. Für den Zeitpunkt t=5 s werden in Abb. 3.11 (rechts) zusätzlich in einer räumlichen Darstellung die Lage der beschädigten Konfiguration und die auf diese einwirkende Druckverteilung als Farbkodierung gezeigt. Die für das beschädigte

Abbildung 3.9: **Konvergenzhistorien der TAU-Simulink-gekoppelten Trimmrechnung für das unbeschädigte FMTA.*Links*: Änderung der residuellen Kräfte und Momente über der Anzahl an Datenaustauschen. *Rechts*: Änderung der Trimmfreiheitsgrade über der Anzahl an Datenaustauschen.**

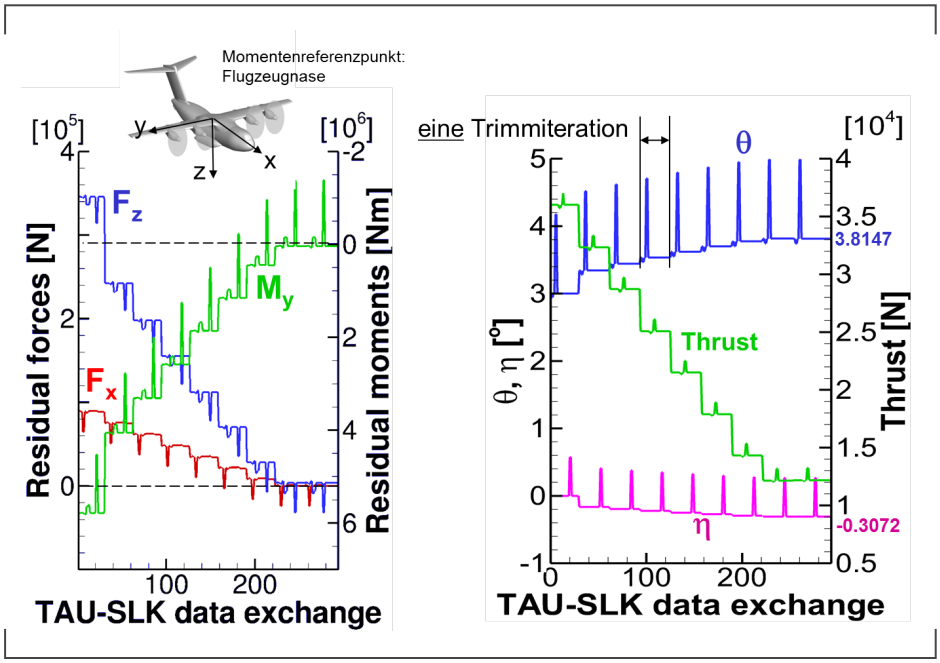


Abbildung 3.10: **Erwartete Bewegung des FMTA infolge der plötzlich eingetretenen Beschädigung am Höhenleitwerk.**

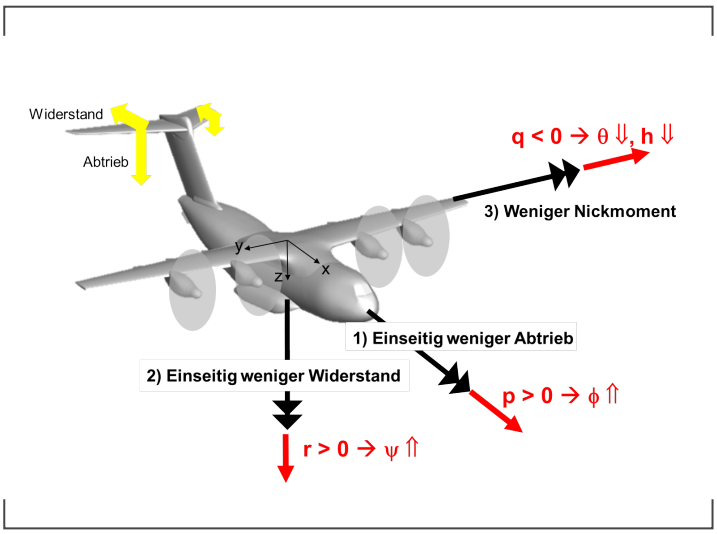
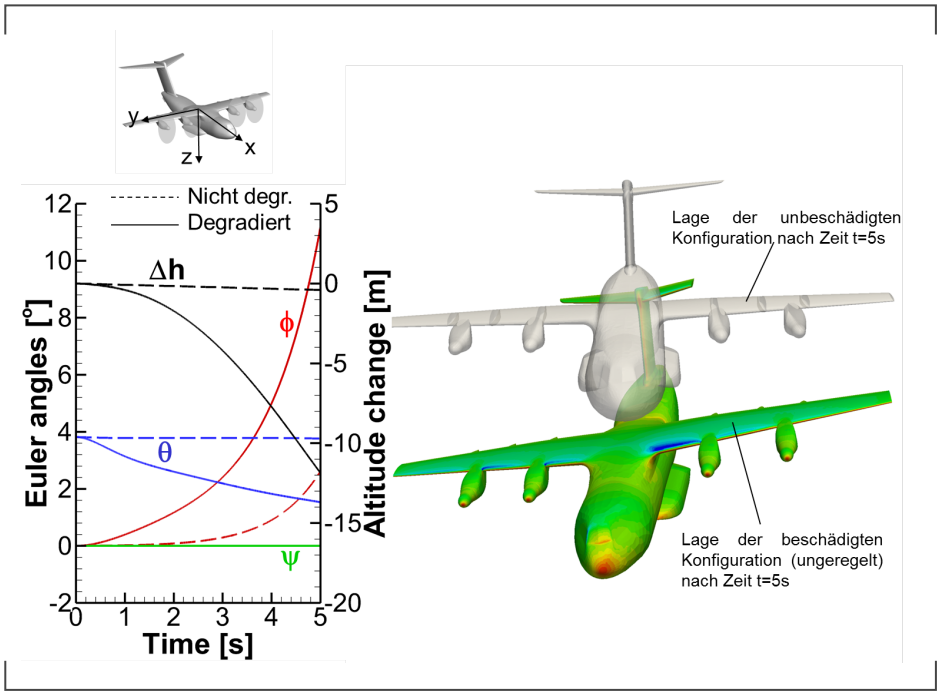


Abbildung 3.11: **Lagewinkeländerungen und Höhenverlust des FMTA infolge des ausgewählten Beschädigungsszenarios (TAU-Simulink-gekoppelte flugdynamische Simulation).**



FMTA vorhergesagte Bewegung stimmt mit der zuvor beschriebenen Erwartungshaltung überein. Das beschädigte FMTA verliert an Höhe h (siehe $h(t)$-Verlauf), reduziert den Nickwinkel θ (siehe $\theta(t)$-Verlauf), beginnt zu rollen (siehe $\phi(t)$-Verlauf) und baut einen positiven Gierwinkel ψ auf. Letzteres ist aufgrund der Achsenskalierung nicht im Diagramm zu sehen.

Da die vorhergehende Trimmrechnung nicht vollständig auskonvergiert wurde — es bestanden noch kleine residuelle Kräfte und Momente in der Größenordnung 10^{-4} —, weicht das unbeschädigte FMTA bei Fortsetzung der Simulation im instationären gekoppelten Modus geringfügig vom Trimmpunkt ab (siehe insbesondere ϕ- und h-Kurve in Abb. 3.11 links). Der Funktionsnachweis der umgesetzten Kopplung kann dennoch als erfolgreich angesehen werden.

4 Zusammenfassung

Im Rahmen dieses Berichts wurde die Kopplung zwischen dem CFD-Löser TAU und dem Simulink-Modell des FMTA vorgestellt, die im Rahmen des Projekts MiTraPor II umgesetzt wurde. Die im Rahmen der Kopplung entwickelte Schnittstelle basierend auf einer Socket-Kommunikation in Python bietet ein hohes Maß an Flexibilität. Sie ermöglicht die Ausführung des CFD-Lösers TAU im parallelen Modus auf üblichen Hochleistungsrechnerarchitekturen. Gekoppelt dazu kann die gleichzeitige Ausführung von Simulink im seriellen Modus auf einem PC erfolgen, der beliebig *außerhalb* des Hochleistungsrechnernetzwerks positioniert sein kann. Die CFD-seitige Kopplung basiert auf der Nutzung des *FlowSimulator*s. Der Funktionsnachweis der Kopplung wurde erfolgreich für ein ausgewähltes Beschädigungsszenario erbracht. Ein und dieselbe Kopplung kann in den Modi *Trimmung* und *Flugdynamische Simulation* gleichermaßen verwendet werden.

Darüber hinaus wurde eine effiziente Funktionalität im *FlowSimulator* zur Verstellung von Steuerflächen auf der Basis von Netzdeformation im Projekt entwickelt. Sie wurde im vorliegenden Bericht ausführlich erläutert.

Die entwickelte CFD-Simulink-Kopplung schafft gegenüber der zuvor verfügbaren reinen Simulink-Modellierung mit vereinfachter derivativbasierter Aerodynamik einen nachhaltigen Mehrwert im Hinblick auf die Analyse- und Bewertungsfähigkeit von militärischen Transportflugzeugen ohne und mit Beschädigung.

A Details zu den CFD-Netzen

Die Verteilung der sogenannten Marker, mittels denen die einzelnen Oberflächen im CFD-Netz idenfiziert werden können — dies ist wichtig im Rahmen der Verstellung der Steuerflächen —, sind für die CFD-Netze ohne und mit Beschädigung aus Abb. 3.2 in der Tabelle A.1 aufgelistet.

Die Lage der Scharnierpunkte (grüne Punkte) und Scharnierachsen (blaue Pfeile), die im Rahmen der Verstellung der Steuerflächen verwendet wurden, sind in Abb. A.1 abgebildet. In Tabelle A.2 sind die Koordinaten der Scharnierpunkte angegeben. Die in Abb. A.1 eingetragenen Pfeilrichtungen geben jeweils positive Ausschlagswinkel im Sinne der Methode zum Verstellen der Steuerflächen in der CFD-Simulation an.

Tabelle A.1: **Verteilung der TAU-Marker in generierten CFD-Netzen.**

Komponente	Marker *unbeschädigte Konfig.*	Marker *beschädigte Konfig.*
Fernfeld	1	1
Rechtes Querruder	22, 47	22, 45
Linkes Querruder	44, 49	42, 47
Seitenruder	6, 28	6, 28
Rechtes Höhenruder	11, 13, 51, 52	11, 13, 49, 50
Linkes Höhenruder	33, 35, 55, 56	—
Gesamte rechte Trimmfläche	8-13, 50-53	8-13, 48-51
Gesamte linke Trimmfläche	30-35, 54-57	30-33, 52-57
Funktionsfläche für Trimmflächenverstellung	23, 45	23, 43

Abbildung A.1: **Angenommene Lagen der Scharnierpunkte und -achsen der Steuerflächen des FMTA in den verwendeten CFD-Netzen.**

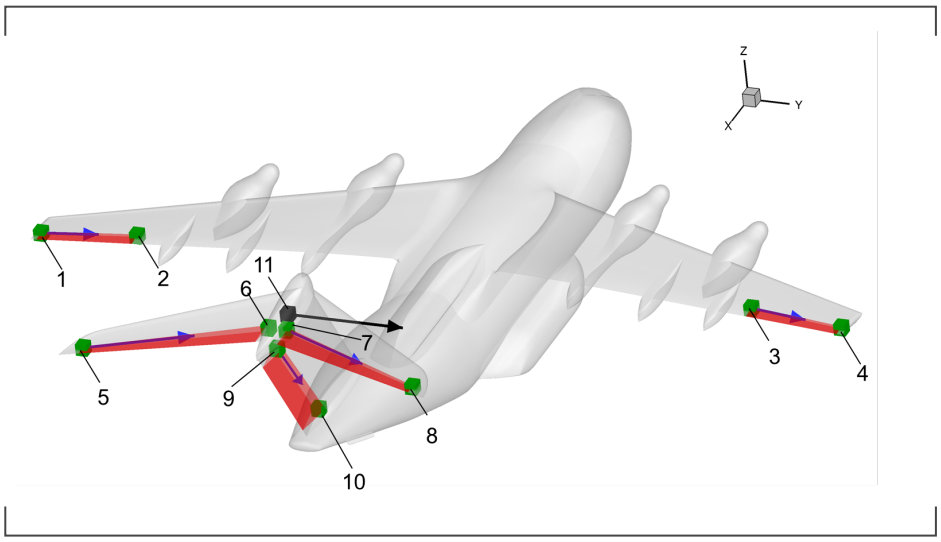


Tabelle A.2: **Koordinaten der Scharnierpunkte der FMTA-Steuerflächen. Die Scharnierpunktnummerierung folgt derjenigen aus Abb. A.1 verwiesen.**

Scharnier-punktnr.	Welcher Komponente zugeordnet ?	x-Koordinate [m]	y-Koordinate [m]	z-Koordinate [m]
1	li. Querruder	21.32592	-20.88471	2.45868
2		20.88471	-16.03119	2.666916
3	re. Querruder	20.88471	16.03119	2.666916
4		21.32592	20.88471	2.45868
5	li. Höhenruder	43.963374	-8.594439	10.358628
6		40.18119	-0.4683	10.327695
7	re. Höhenruder	40.18119	0.4683	10.327695
8		43.963374	8.594439	10.358628
9	Seitenruder	40.477101	0.	9.419382
10		36.56247	0.	3.975363
11	Trimmfläche	38.854263	0.	10.382694

B Definition auszutauschender Kopplungsgrößen

In den Tabellen B.1 und B.2 sind die Größen aufgelistet, die im Rahmen der Kopplung zwischen TAU und Simulink bzw. zwischen der Python-Schicht, die Simulink umgibt (siehe Kap. 2.2), und Simulink ausgetauscht werden. Der Austausch erfolgt dateibasiert. Die Ausgabegrößen der Flugdynamik- bzw. Trimmsimulation werden von Simulink in der in Tab. B.1 angegebenen Reihenfolge in die Datei *slk_output.txt* geschrieben. Die Eingangsgrößen werden von Simulink in der Reihenfolge, wie sie in Tab. B.2 angegeben ist, aus der Datei *slk_input.txt* gelesen. In beiden Fällen werden die Daten durch Leerzeichen getrennt in die erste Zeile der jeweiligen Datei geschrieben. Die Kopplungsgrößen, speziell die vorhanden Steuerflächen, sind auf die FMTA-Konfiguration angepasst. Für den Fall, dass die bestehende Kopplung auf Konfigurationen mit zusätzlich vorhandenen Steuerflächen angewendet werden soll, so muss die bestehende Kopplung um die jeweiligen Größen zunächst erweitert werden.

Tabelle B.1: **Austauschgrößen, die von Simulink zur Übermittlung an den CFD-Prozess in der angegebenen Reihenfolge in die Datei *slk_output.txt* geschrieben werden.**

	Größe	Variable	KOS	Einheit	Kommentar
1	Rollrate	p_{kb}	Abb. E.2	rad/sec	
2	Nickrate	q_{kb}	Abb. E.2	rad/sec	
3	Gierrate	r_{kb}	Abb. E.2	rad/sec	
4	Rollwinkel	ϕ		rad	Winkel vom geodätischem System 1 ins körperfeste Systems 2
5	Nickwinkel	θ		rad	Winkel vom geodätischem System 1 ins körperfeste Systems 2
6	Gierwinkel	ψ		rad	Winkel vom geodätischem System 1 ins körperfeste Systems 2
7	x-Geschwindigkeit	u_{kg}	Abb. E.1	m/sec	

Fortsetzung auf nächster Seite

Tabelle B.1: **(Fortsetzung)**

	Größe	Variable	KOS	Einheit	Kommentar
8	y-Geschwindigkeit	v_{kg}	Abb. E.1	m/sec	
9	z-Geschwindigkeit	w_{kg}	Abb. E.1	m/sec	
10	x-Position	x_0	Abb. E.5	m	x-Position des körperfesten System 2 im geodätischen System 1
11	y-Position	y_0	Abb. E.5	m	y-Position des körperfesten System 2 im geodätischen System 1
12	z-Position	z_0	Abb. E.5	m	z-Position des körperfesten System 2 im geodätischen System 1
13	Querruderausschlag links	ξ_l		rad	Auschlag des rechten Ruders nach unten positiv definiert
14	Querruderausschlag rechts	ξ_r		rad	Auschlag des rechten Ruders nach unten positiv definiert
15	Seitenruderausschlag	ζ		rad	Ausschlag nach links positiv definiert
16	Höhenruderausschlag links	η_l		rad	Ausschlag nach unten positiv definiert
17	Höhenruderausschlag rechts	η_r		rad	Ausschlag nach unten positiv definiert
18	Trimmflossen-ausschlag	η_{HTP}		rad	wie Höhenruder, Ausschlag nach unten positiv definiert
19	Klappenausschlag	η_{Flaps}		rad	
20	Spoilerausschlag rechts	η_{S1R}		rad	innerster Spoiler
21	Spoilerausschlag rechts	η_{S2R}		rad	Spoiler innen mittig
22	Spoilerausschlag rechts	η_{S3R}		rad	Spoiler außen mittig
23	Spoilerausschlag rechts	η_{S4R}		rad	äußerster Spoiler
24	Spoilerausschlag links	η_{S1L}		rad	innerster Spoiler
25	Spoilerausschlag links	η_{S2L}		rad	Spoiler innen mittig
26	Spoilerausschlag links	η_{S3L}		rad	Spoiler außen mittig
27	Spoilerausschlag links	η_{S4L}		rad	äußerster Spoiler
28	Schub	F_1	Abb. E.2	N	linkes äußeres Triebwerk
29	Schub	F_2	Abb. E.2	N	linkes inneres Triebwerk
30	Schub	F_3	Abb. E.2	N	rechtes inneres Triebwerk
31	Schub	F_4	Abb. E.2	N	rechtes äußeres Triebwerk
32	x-Versatz	$b_{TAU,x}$	Abb. E.3	m	Versatz des Systems 4 gegenüber 3 in x-Richtung
33	y-Versatz	$b_{TAU,y}$	Abb. E.3	m	Versatz des Systems 4 gegenüber 3 in y-Richtung

Fortsetzung auf nächster Seite

Tabelle B.1: **(Fortsetzung)**

	Größe	Variable	KOS	Einheit	Kommentar
34	z-Versatz	$b_{TAU,z}$	Abb. E.3	m	Versatz des Systems 4 gegenüber 3 in z-Richtung
35	Zeit	t		sec	Simulationszeit von Simulink
36	Zeitschrittweite	Δt		sec	Zeitschrittweite von Simulink

Tabelle B.2: **Austauschgrößen, die von Simulink aus der Datei *slk_input.txt* in der angegebenen Reihenfolge gelesen werden.**

	Größe	Variable	KOS	Einheit	Kommentar
1	x-Kraft	$R_{Aero,x,b}$	Abb. E.2	N	
2	y-Kraft	$R_{Aero,y,b}$	Abb. E.2	N	
3	z-Kraft	$R_{Aero,z,b}$	Abb. E.2	N	
4	x-Moment	$Q_{Aero,x,b}$	Abb. E.2	Nm	
5	y-Moment	$Q_{Aero,y,b}$	Abb. E.2	Nm	
6	z-Moment	$Q_{Aero,z,b}$	Abb. E.2	Nm	
7	Schallgeschwindigkeit	a		m/sec	
8	Lufttemperatur	T		K	
9	Luftdichte	ϱ		kg/m^3	
10	Luftdruck	p		N/m^2	

C Socket-Kommunikationsschicht

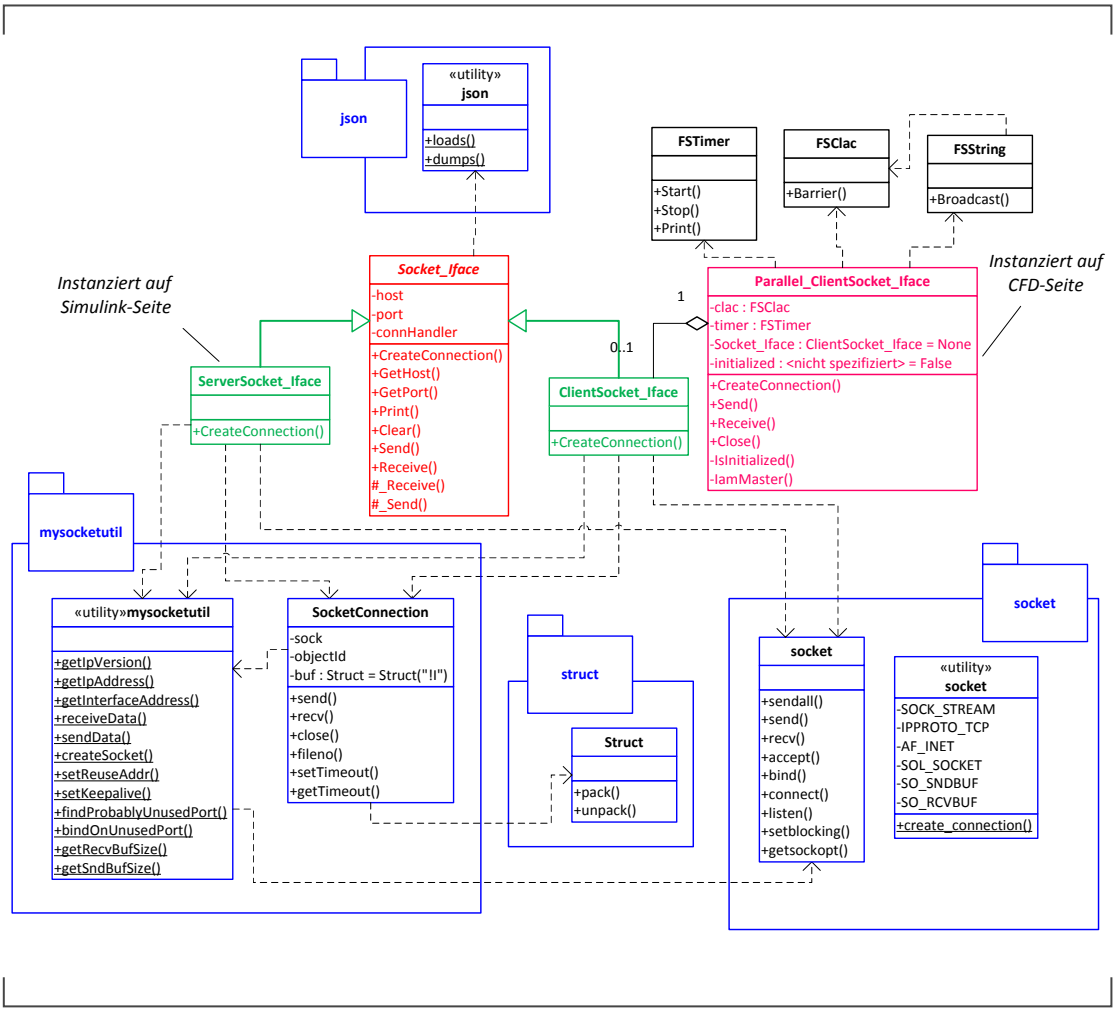
Den grundsätzlichen Aufbau der Socket-Kommunikationsschicht (UML-Diagramm der Klassenbeziehungen) zeigt Abb. C.1. Ausgehend von der Basisklasse *Socket_Iface* werden die Klassen *ServerSocket_Iface* (erstellt einen Socket-Server) und *ClientSocket_Iface* (erstellt einen Socket-Client) abgeleitet. Sie implementieren jeweils die Methode *CreateConnection*, die eine Verbindung zu einem anderen Socket etabliert. Auf Simulink-Seite wird eine Instanz der Klasse *ServerSocket_Iface* erzeugt, auf CFD-Seite eine Instanz der Klasse *Parallel_ClientSocket_Iface* (siehe rechten Teil der Abb. C.1). Innerhalb des Konstruktors von *Parallel_ClientSocket_Iface* wird vom *Master*-Prozess (Prozess 0) — und ausschließlich von diesem — eine Instanz von *ClientSocket_Iface* gebildet. *Parallel_ClientSocket_Iface* nutzt desweiteren *FlowSimulator*-Funktionalitäten (*FSClac*, *FSString*), um beim Senden und Empfangen von Daten den CFD-seitig bestehenden parallelen Kontext zu berücksichtigen (siehe hierzu Kap. C.3).

Die Klassen *ServerSocket_Iface* und *ClientSocket_Iface* nutzen innerhalb der Methode *CreateConnection* die Basisfunktionalitäten, die von den Python-Modulen *mysocketutil* und *socket* bereitgestellt werden. Die im Modul *mysocketutil* implementierte Klasse *SocketConnection* (siehe Programmausdruck C.4) nutzt das Python-Modul *struct* zur Erzeugung eines Integer-Puffers.

Innerhalb von *Socket_Iface* wird zur Serialisierung und Entserialisierung der Daten im Rahmen der Sende- und Empfangsoperationen das Python-Modul *json* genutzt (siehe Kap. C.3).

Die implementierte Socket-Schnittstelle erzeugt Stream-Sockets im IPv4-Adressraum, die dem TCP-Protokoll folgen und Sende- und Empfangsoperationen im *Blocking*-Modus durchführen.

Abbildung C.1: **UML-Klassendiagramm der Python-basierten Schnittstelle zur Socket-basierten Kommunikation zwischen CFD- und Simulink-Seite.**



C.1 Erzeugung eines Socket-Servers (Simulink-Seite)

Der Programmausdruck C.1 zeigt die Implementierung der Methode *CreateConnection* der *ServerSocket_Iface*-Klasse. In Zeile 6 wird ein Socket-Objekt vom gewünschten Typ (Stream-Socket, TCP-Protokoll, IPv4-Adressraum) erzeugt. In Zeile 8 wird dieses Socket-Objekt an die gewünschte IP-Adresse und den Rechner-Port gebunden. IP-Adresse und Port wurde dem Konstruktor von *ServerSocket_Iface* als Argumente übergeben. Nach der Bindung wartet der Server-Socket auf Verbindungsanfragen von etwaigen Clients (Z. 9). Im vorliegenden Fall wird nur eine Verbindung akzeptiert. Weitere Anfragen werden abgewiesen. In Zeile 10 wird die Verbindung zum anfragenden Client aufgebaut. Die Methode *accept* liefert ein neues Socket-Objekt (Variable *conn*) zurück, das fortan die Verbindung repräsentiert. Es wird innerhalb einer Instanz der Klasse *SocketConnection* aus dem *mysocketutil*-Modul gespeichert (Z. 11). An *SocketConnection* werden jeweilig vorhandene Aufrufe von Sende- und Empfangsmethoden delegiert (siehe Kap. C.3). Eine Instanz von *SocketConnection* wird als Membervariable (*connHandler*) der Klasse *Socket_Iface* gespeichert.

Programmausdruck C.1: **Erzeugung eines Server-Sockets in *ServerSocket_Iface*.**

```
1  import socket
2  import mysocketutil
   class ServerSocket_Iface(Socket_Iface)
4      ...
       def CreateConnection(self):
6          sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)
7          sock.setblocking(True)
8          sock.bind((self.GetHost(), self.GetPort()))
9          sock.listen(1)
10         conn, addr = sock.accept()
11         self.connHandler = mysocketutil.SocketConnection(conn)
```

C.2 Erzeugung eines Socket-Clients (CFD-Seite)

Der Programmausdruck C.2 demonstriert, wie innerhalb der Methode *CreateConnection* der Klasse *ClientSocket_Iface* eine Socketverbindung als Client aufgebaut wird. Zeile 6 generiert eine Socket-Objekt des gewünschten Typs (Stream-Socket, TCP-Protokoll, IPv4-Adressraum). In Zeile 8 wird eine Verbindungsanfrage an den angegebenen Server-Socket gestellt. Ist diese erfolgreich, so wird ein neues Socket-Objekt (Variable *conn*) zurückgege-

ben und dieses genauso wie im Programmausdruck C.1 zur weiteren Verwaltung an die Instanz der Klasse *SocketConnection* weitergegeben.

Programmausdruck C.2: **Erzeugung eines Client-Sockets in *ClientSocket_Iface*.**

```
import socket
import mysocketutil
class ClientSocket_Iface(Socket_Iface)
    ...
    def CreateConnection(self):
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)
        sock.setblocking(True)
        conn = socket.create_connection((self.GetHost(), self.GetPort()))
        conn.settimeout(None)
        self.connHandler = mysocketutil.SocketConnection(conn)
```

C.3 Durchführung von Sende- und Empfangsoperationen

Im Folgenden wird der Programmablauf bei Aufruf der Methoden *Send* und *Receive* erläutert, die als Bestandteil der Basisklasse *Socket_Iface* implementiert sind (siehe Programmausdruck C.3) und als solches an *ServerSocket_Iface* und *ClientSocket_Iface* vererbt werden. Ausschließlich diese beiden Top-Level-Methoden werden von den Prozessskripten *run_CFD.py* und *run_FM.py* (siehe Programmausdrucke 2.5 (S. 22)) und 2.7 (S. 27) zum Senden und Empfangen von Daten über die Socket-Schnittstelle verwendet.

Programmausdruck C.3: **Serielle Top-Level-Methoden zum Senden (*Send*) und Empfangen (*Receive*) von Daten über die Socket-Schnittstelle.**

```
class Socket_Iface:
    ...
    def Send(self, data):
        buf = json.dumps(data)
        self._Send(buf)

    def Receive(self):
        buf = self._Receive()
        data = json.loads(buf)
        return data

    def _Send(self, data):
        self.connHandler.send(data)

    def _Receive(self):
        return self.connHandler.recv()
```

Die *Send*- und *Receive*-Methoden von *Socket_Iface* nutzen das Python-Modul *json*, um Daten vor dem Senden zu serialisieren bzw. sie nach dem Empfangen zu entserialisieren. *Socket_Iface* ruft die Sende- und Empfangsmethoden der Klasse *SocketConnection* auf (Zeilen 13 und 16 des Programmausdrucks C.3), um den Sende- und Empfangsvorgang durchzuführen. Die Klasse *SocketConnection* — inklusive dessen Methoden *send* und *recv* — wird im Programmausdruck C.4 gezeigt.

Programmausdruck C.4: **Klasse *SocketConnection* aus dem Python-Modul *mysocketutil*. Methoden *send* und *recv* realisieren Senden und Empfangen von Daten über die Socket-Verbindung *sock* unter Verwendung der Funktionen *sendData* und *receiveData* aus *mysocketutil*.**

```
1  class SocketConnection(object):
2      RECEIPT = 1
       def __init__(self, sock):
4          self.sock     = sock
           self.buf      = struct.Struct("!I")
6
       def send(self, msg):
8          msglenPacked = self.buf.pack(len(msg)) # Verpacke Nachrichtenlaenge
9          sendData(self.sock, msglenPacked) # Sende gep. Nachrichtenlaenge
10         sendData(self.sock, msg) # Sende Nachricht
11         receiptPacked = receiveData(self.sock, self.buf.size) # Empf. Empfangsbest.
12         receipt = self.buf.unpack(receiptPacked)[0] # Entpacke Empfangsbestaetigung
           if receipt != self.RECEIPT:
14             raise Error("Did not receive receipt !!!", __file__)
16     def recv(self):
           msglenPacked = receiveData(self.sock, self.buf.size) # Empf. gep. Nachrichtenl.
18         msglen = self.buf.unpack(msglenPacked)[0] # Entpacke Nachrichtenlaenge
           msg = receiveData(self.sock, msglen) # Empf. Nachricht
20         receiptPacked = self.buf.pack(self.RECEIPT) # Verpacke Empfangsbestaetigung
           sendData(self.sock, receiptPacked) # Sende verpackte Empfangsbestaetigung
22         return msg
```

SocketConnection realisiert die Sende- und Empfangsoperationen wie folgt. Als erstes wird jeweils die Länge der zu sendenden eigentlichen Nachricht (d. h. die Länge der Zeichenkette) gesendet (Z. 9 im Programmausdruck C.4). Die Länge wird in einem Puffer der Größe 4 Byte (Nutzung der Klasse *Struct* aus dem Python-*struct*-Modul) zunächst binär verpackt (Z. 8) und der Pufferinhalt versendet. Bei der nötigen Empfangsoperation ist die Menge der zu empfangenden Daten auf Empfängerseite also vorab bekannt. Die eigentlichen Daten werden danach versendet (Z. 10). Der Empfänger kennt nun genau die Menge der zu erwarteten Daten und kann daraufhin, die Empfangsoperation gestalten — es müssen evtl. mehrere Empfangsoperationen durchgeführt werden bis die volle Menge der erwarteten Daten empfangen wurde. Danach quittiert der Empfänger dem Sender den erfolgreichen Empfang der Daten durch Senden eines 4 Byte-Integers (Z. 21). Der Sender, der die Sende-

operation ursprünglich initiiert hat, wartet solange, bis diese Bestätigung eintrifft (Z. 11). Jeder Sende-Empfangsvorgang stellt dadurch einen Synchronisationspunkt dar, der sicherstellt, dass beidseitig der Kommunikationsvorgang vollständig abgeschlossen wurde.

Programmausdruck C.5: **Low-Level-Methode aus dem Python-Modul *mysocketutil* zum Senden der Daten *data* über die Socketverbindung *sock* mittels des Python-Pakets *socket*.**

```
1  def sendData(sock, data):
2      """Send some data over a socket. Some systems have problems with ''sendall()'' when
        the socket is in non-blocking mode, e.g., Mac OS X seems to be happy to throw
       EAGAIN errors too often. This function falls back to using a regular send loop if
       needed."""
       if sock.gettimeout() is None:
4          # Socket is in BLOCKING mode, we can use sendall normally.
           while True:
6              try:
                   ret = sock.sendall(data)
8                  if ret is None:
                       return
10                 else:
                       raise Error("Sendall returned without success !", __file__)
12             except socket.timeout:
                   raise Error("sending: timeout", __file__)
14             except socket.error:
                   x=sys.exc_info()[1]
16                 err=getattr(x, "errno", x.args[0])
                   raise Error("sending: connection lost: %s"%str(x), __file__)
18
19     else:
20         # Socket is in NON-BLOCKING mode, use regular send loop.
           retrydelay=0.0
22         iter = 0
           while data:
24             try:
                   sent = sock.send(data)
26                 data = data[sent:]
               except socket.timeout:
28                 raise Error("sending: timeout", __file__)
               except socket.error:
30                 x=sys.exc_info()[1]
                   err=getattr(x, "errno", x.args[0])
32                 if err not in ERRNO_RETRIES:
                       raise Error("sending: connection lost: %s" %str(x), __file__)
34                 # add a slight delay to wait before retrying
                   time.sleep(0.00001+retrydelay)
36                 retrydelay=__nextRetrydelay(retrydelay)
```

Beim Senden und Empfang bedient sich die Klasse *SocketConnection* den Funktionen *sendData* und *receiveData* aus dem *mysocketutil*-Modul. Beide Funktionen werden in den Programmausdrucken C.5 und C.6 gezeigt.

Im *Blocking*-Modus, der hier genutzt wird, verwendet die Funktion *sendData* die Methode *sendall* aus dem *socket*-Modul (Zeilen 3-18 im Programmausdruck C.5). Im Gegensatz zur Basismethode *send* stellt diese definitiv sicher, dass alle Daten wirklich versendet wurden.

Die *receiveData*-Funktion unterscheided, ob die Empfangsoption *MSG_WAITALL* auf der jeweilig verwendeten Rechnerarchitektur vorhanden ist. Ist dies der Fall (Z. 18-47 im Programmausdruck C.5), so kann auf eine Schleife mit mehreren Low-Level-Aufrufen von *socket.recv()* verzichtet werden, da die Option sicherstellt, dass in jedem Fall die Daten in der geforderten Menge empfangen wurden. Unter dem Windows-Betriebssystem — unter diesem lief Simulink im vorliegenden Fall — funktioniert diese Option allerdings nicht. In dem Fall ruft *receiveData* solange *socket.recv()* auf (Z. 49-82), bis die geforderte Menge an Daten wirklich empfangen wurde.

Programmausdruck C.6: **Low-Level-Methode aus dem Python-Modul *mysocketutil* zum Empfangen einer Datenmenge *size* über die Socketverbindung *sock* mittels des Python-Pakets *socket*.**

```
1  if sys.platform=="win32":
2      # it doesn't work reliably on Windows even though it's defined
       USE_MSG_WAITALL = False
4  else:
       USE_MSG_WAITALL = hasattr(socket, "MSG_WAITALL")
6
   def receiveData(sock, size):
8      """Retrieve a given number of bytes from a socket.
       It is expected the socket is able to supply that number of bytes.
10     If it isn't, an exception is raised (you will not get a zero length result
       or a result that is smaller than what you asked for). The partial data that
12     has been received however is stored in the 'partialData' attribute of
       the exception object."""
14     try:
           retrydelay=0.0
16         msglen=0
           chunks=[]
18         if USE_MSG_WAITALL:
19             # Waitall is very convenient. if a socket error occurs,
20             # we can assume the receive has failed. No need for a loop,
               # unless it is a retryable error.
22             # Some systems have an erratic MSG_WAITALL and sometimes still return
               # less bytes than asked. In that case, we drop down into the normal
24             # receive loop to finish the task.
               iter=0
26             while True:
                   try:
28                     data=sock.recv(size, socket.MSG_WAITALL)
                       if len(data)==size:
30                         return data
                       # less data than asked, drop down
32                     # into normal receive loop to finish
                       msglen=len(data)
```

```
34                      chunks=[data]
                        break
36                  except socket.timeout:
                        raise Error("receiving: timeout", __file__)
38                  except socket.error:
                        x=sys.exc_info()[1]
40                      err=getattr(x, "errno", x.args[0])
                        print err
42                      if err not in ERRNO_RETRIES:
                            raise Error("receiving: connection lost: %s"%str(x), __file__)
44                      # add a slight delay to wait before retrying
                        time.sleep(0.00001+retrydelay)
46                      retrydelay=__nextRetrydelay(retrydelay)
                    iter += 1
48          # old fashioned recv loop, we gather chunks until the message is complete

50          iter = 0
            subiter = 0
52          while True:
                try:
54                  while msglen < size:
                        # 60k buffer limit avoids problems on certain OS, e.g.,
56                      # VMS, Windows
                        chunk=sock.recv(min(60000, size-msglen))
58                      if not chunk:
                            break
60                      chunks.append(chunk)
                        msglen+=len(chunk)
62                      subiter += 1
                    data=EMPTY_BYTES.join(chunks)
64                  del chunks
                    if len(data)!=size:
66                      errmsg = "receiving: not enough data. "+\
                                "data received so far <%s>" %str(data)
68                      raise Error(errmsg, __file__)
                    return data  # yay, complete
70              except socket.timeout:
                    raise Error("receiving: timeout", __file__)
72              except socket.error:
                    x=sys.exc_info()[1]
74                  err=getattr(x, "errno", x.args[0])
                    if err not in ERRNO_RETRIES:
76                      raise Error("receiving: connection lost: %s" %str(x), __file__)
                    else:
78                  # add a slight delay to wait before retrying
                    time.sleep(0.00001+retrydelay)
80                  retrydelay=__nextRetrydelay(retrydelay)
                iter += 1
82
83      except socket.timeout:
            raise Error("receiving: timeout", __file__)
84
```

C.3.1 Senden und Empfangen im parallelen Kontext

Auf der CFD-Seite muss der dort bestehende parallele Kontext beim Senden und Empfangen von Daten über die Socket-Schnittstelle berücksichtigt werden. Daher werden dort die Methoden *Send* und *Receive* der Klasse *Parallel_ClientSocket_Iface* genutzt (siehe Programmausdruck C.7). Auf dem *Master*-Prozess (Prozess 0) erzeugt *Parallel_ClientSocket_Iface* eine Instanz der Klasse *Socket_Iface*, welche ein serielles Socket-Interface (siehe Programmausdruck C.3) implementiert.

Beim *parallelen* Senden (Z. 3ff. des Programmausdrucks C.7) sendet nur der *Master*-Prozess während die anderen Prozesse warten bis dieser Vorgang abgeschlossen ist.

Beim entsprechenden Empfangen (Z. 8ff.) empfängt ebenfalls nur der *Master*-Prozess die gesendeten Daten. Die empfangenen Daten werden anschließend vom *Master* unter Verwendung der *FSDM*-Objekte *FSString* und *FSClac* an alle anderen Prozesse gesendet.

Programmausdruck C.7: **Parallele Top-Level-Methoden zum Senden (*Send*) und Empfangen *Receive*) von Daten über die Socket-Schnittstelle auf CFD-Seite.**

```
1  class Parallel_ClientSocket_Iface:
2      ...
3      def Send(self, data):
4          if self.clac.ProcID() == 0:
5              self.Socket_Iface.Send(data)
6          self.clac.Barrier()
7
8      def Receive(self):
9          fsStr = FSString()
10         if self.clac.ProcID() == 0:
11             data = self.Socket_Iface.Receive()
12             fsStr = FSString(str(data))
13         self.clac.Barrier()
14         fsStr.Broadcast(self.clac, 0)
15         pyStr = str(fsStr)
16         try:
17             data = eval(pyStr)
18         except:
19             data = pyStr
20         return data
```

D Erstellte *FlowSimulator*-Hilfsfunktionalitäten

D.1 Handhabung von TAU-Markerlisten

Im Rahmen der Netzdeformation *FSDeformation* werden Randbedingungen mit den Oberflächennetzsegmenten assoziiert. Die Oberflächennetzsegmente werden über TAU-Marker (im *FSDM*-Kontext *CADGroupIDs*) identifiziert. Sie müssen als Python-Listen an *FSDeformation* übergeben werden. Häufig sind die Oberflächennetze stark segmentiert, d. h. bestehen aus vielen Markern. Mittels der Klasse *MarkerHandler* (siehe Programmauszug D.1) lassen sich einfacher lange Listen von Markern angeben. Der Programmauszug D.2) zeigt für den konkreten Fall des FMTA die Nutzung von *MarkerHandler*. Bspw. liefert die Zeile 2 basierend auf der gegebenen Zeichenkette für die Variable *markersHtpR* die Python-Liste `[8,9,10,11,12,13,50,51,52,53]` zurück.

Programmausdruck D.1: **Klassendeklaration für bequeme Handhabung von TAU-Markerlisten.**

```
1  class MarkerHandler:
2
3      def CreateMarkerList(self, markerString):
4          """
5          returns a Python list from a comma-separated string which may include also
6          interval instructions
7          """
8
9          mList = []
10
11         if len(markerString) > 0:
12             mStringList = markerString.split(",")
13
14             for m in mStringList:
15                 nMinusSigns = m.count("-")
16                 if nMinusSigns == 0:
17                     mList.append( int(m) )
18                 elif nMinusSigns == 1:
19                     mm = m.split("-")
20                     for i in range(int(mm[0]), int(mm[1])+1):
```

```
                        mList.append( i )
22                  else:
                        FSError.SetAndExit("Found more than 1 minus sign " +\
24                                         "in marker interval statement <%s>" %m)
            return mList
26
        def GetSelectionForListOfMarkers(self, listOfMarkers):
28          """
            for a given Python list with marker IDs this function returns
30          the corresponding mesh selection
            """
32          selection = FSMeshSelectionOpAttribute(FSMeshEnums.CT_Undefined,
                                                   FSString(FS_AT_CADGroupID),
34                                                 listOfMarkers[0])

36          for m in listOfMarkers[1:]:
                selection = selection |
38                          FSMeshSelectionOpAttribute(FSMeshEnums.CT_Undefined,
                                                       FSString(FS_AT_CADGroupID),
40                                                     m)
            return selection
```

Programmausdruck D.2: **Erzeugung von Python-Listen aus Intervalangaben zu TAU-Markern mit der Klasse aus Programmausdruck D.1.**

```
1 markerHandler = MarkerHandler()
markersWholeAC          = markerHandler.CreateMarkerList("2-57")
3 markersAileronR         = markerHandler.CreateMarkerList("22,47")
markersAileronL         = markerHandler.CreateMarkerList("44,49")
5 markersRudder           = markerHandler.CreateMarkerList("6,28")
markersElevatorR        = markerHandler.CreateMarkerList("11,13,51,52")
7 markersElevatorL        = markerHandler.CreateMarkerList("33,35,55,56")
markersHtpR             = markerHandler.CreateMarkerList("8-13,50-53")
9 markersHtpL             = markerHandler.CreateMarkerList("30-35,54-57")
markersHtpRotationSurf = markerHandler.CreateMarkerList("23,45")
```

Ferner erleichert die Methode *GetSelectionForListOfMarkers* der Klasse *MarkerHandler* die Erzeugung von *FSMeshSelection*-Operationsobjekten. Dies zeigt Programmausdruck D.3. In Zeile 2 wird für ein gegebenes *FSMesh*-Objekt (hier: das Volumennetz des FMTA) ein neues *FSMesh*-Objekt zurückgeben (hier: das Oberflächennetz des FMTA exklusive Fernfeld), das nur den selektierten Teil enthält.

Programmausdruck D.3: **Nutzung der Klasse aus Programmausdruck D.1 zur Erzeugung einer Netzselektion.**

```
1 SurfSelection = markerHandler.GetSelectionForListOfMarkers(markersWholeAC)
2 SurfMesh = surfSelection.Apply(CFDMesh)
```

D.2 Berechnung des Verschiebungsfeldes zu einer Starrkörperrotation für eine Gruppe von Oberflächennetzsegmenten

Zur Verstellung der Trimmflosse mittels Netzdeformation muss an *FSDeformation* ein enstprechendes Verschiebungsfeld als Randbedingung übergeben werden (siehe Erläuterungen in Kap. 3.3.1). Die im Programmausdruck D.4 gezeigte Klasse *SurfMeshRotationHandler* dient dazu, ein solches Verschiebungsfeld unter Vorgabe von Rotationsachse (Urpsprung und Achsenvektor), Rotationswinkel für eine Gruppe von Oberflächennetzsegmenten (identifiziert durch TAU-Marker) bequem zu berechnen. Für ein Volumennetz, das im *FSDataManager* unter einem bestimmten Namen adressierbar ist, muss zunächst das Oberflächennetz, auf das die Rotation wirken soll, mit der Methode *GetAndRegisterSurfaceMesh* ebenfalls unter einen Namen im *FSDataManager* registriert werden. Anschließend kann mit der Methode *GetDefoVectorsForRotation* ein Verschiebungsfeld berechnet werden (siehe Programmausdrucke 3.2 (S. 39) und 3.4 (S. 41)). *GetDefoVectorsForRotation* gibt die Oberflächenpunkte und die Verschiebungsvektoren als *NumPy*-Arrays der Dimension $(n, 3)$ zurück.

Die Klasse nutzt u. a. das *FSNumPyInterface*-Plugin und die *MarkerHandler*-Klasse aus Programmausdruck D.1.

Programmausdruck D.4: **Klassendeklaration zur Vorgabe eines Verschiebungsfeldes für *FSDeformation*, das eine Starrkörperrotation von Oberflächennetzanteilen repräsentiert. Vorzugeben sind lediglich Rotationswinkel und Rotationsachse.**

```
1  class SurfMeshRotationHandler:
2      """
       used to get base points and displacements for a rigid-body rotation of surface mesh
        patches
4      """
       def __init__(self, dataManager):
6
           import FSNumPyInterface
8          from FSNumPyArray import FSNumPyArray

10         self.dataManager = dataManager
           self.clac = dataManager.GetClac()
12         self.arrayConverter = FSNumPyArray(self.clac)
           self.markerHandler = MarkerHandler()
14
       # Required function arguments:
16     # MeshKey: String
       # SurfMeshKey: String representing the surface mesh in the data manager
18     # CADGroupIDs: Python list with integer values representing marker IDs
```

```
    def GetAndRegisterSurfaceMesh(self, **kwargs):
        mesh = self.dataManager.GetMesh(kwargs["MeshKey"])
        surfMesh = self.dataManager.GetMesh(kwargs["SurfMeshKey"])
        selection = self.markerHandler.GetSelectionForListOfMarkers(kwargs["CADGroupIDs
    "])
        selection.ApplyInto(surfMesh, mesh)
        return surfMesh

    # Required function arguments:
    # HingePoint: Python list of 3 floats representing the location of the hinge point
    # Angle: Python float representing the rotation angle in degrees
    # Axis: Python character "X", "Y" or "Z" representing the axis of rotation
    # SurfMeshKey: String representing the surface mesh in the data manager
    def GetDefoVectorsForRotation(self, **kwargs):

        surfMesh = self.dataManager.GetMesh(kwargs["SurfMeshKey"])
        surfMesh.CreateGlobalNumbering()
        xyz = self.__GetCoordinatesFromMesh(surfMesh)

        self.__RotateMesh(SurfMeshKey=kwargs["SurfMeshKey"],
                          HingePoint=kwargs["HingePoint"],
                          Axis=kwargs["Axis"],
                          Angle=kwargs["Angle"])

        # TransformMesh seems to switch to local numbering ...
        surfMesh.CreateGlobalNumbering()
        xyzRotated = self.__GetCoordinatesFromMesh(surfMesh)

        # Rotate back
        self.__RotateMesh(SurfMeshKey=kwargs["SurfMeshKey"],
                          HingePoint=kwargs["HingePoint"],
                          Axis=kwargs["Axis"],
                          Angle=-kwargs["Angle"])

        xyzDelta = xyzRotated - xyz
        basePoints = self.arrayConverter.Convert2FSFloatArray(xyz)
        defoVecs   = self.arrayConverter.Convert2FSFloatArray(xyzDelta)
        basePoints.Gather(self.clac, 0)
        defoVecs.Gather(self.clac, 0)
        return basePoints, defoVecs

    # PRIVATE METHODS ------------------------------------------------------------
    # Required function arguments:
    # HingePoint: Python list of 3 floats representing the location of the hinge point
    # Angle: Python float representing the rotation angle in degrees
    # Axis: Python character "X", "Y" or "Z" representing the axis of rotation
    # SurfMeshKey: String representing the surface mesh in the data manager
    def __RotateMesh(self, **kwargs):
        transforms = ( \
            ("Translate",{"DeltaX" : -kwargs["HingePoint"][0],
                          "DeltaY" : -kwargs["HingePoint"][1],
                          "DeltaZ" : -kwargs["HingePoint"][2]}),
            ("RotateAroundAxis", {"Angle" : "%s [deg]" % (str(kwargs["Angle"])),
                                  "Axis"  : tuple(kwargs["Axis"])}),
            ("Translate",{"DeltaX" : kwargs["HingePoint"][0],
```

```
                            "DeltaY" : kwargs["HingePoint"][1],
74                          "DeltaZ" : kwargs["HingePoint"][2]}))

76          meshOps = (("TransformMesh", {"Transformations" : transforms}),)
            surfMesh = self.dataManager.GetMesh(kwargs["SurfMeshKey"])
78
            if not surfMesh.DoOps(meshOps):
80              FSError.PrintAndExit()

82      def __GetCoordinatesFromMesh(self, mesh):
            coordinates = mesh.GetUnstructDataset(str(FSDataName.Coordinates()))
84          xyzValues = coordinates.GetValues()
            xyzValuesNumPy = self.arrayConverter.Convert2Numpy(xyzValues)
86          return xyzValuesNumPy
```

D.3 *Get/Set*-Operation für *FSDatasets* in *FSMesh*-Objekten

Die Klasse *UnstructDatasetHandler* aus Programmausdruck D.5 ermöglicht basierend auf dem gegebenen Datensatznamen das Extrahieren eines Datensatzes als *NumPy*-Array (*Get*-Methode) bzw. das Ersetzen eines Datensatzes mit einem übergebenen *NumPy*-Array (*Set*-Methode). Die Nutzung der Methode wird in den Programmausdrucken 3.1, 3.4, 3.7 und 3.8 (S. 39ff.) gezeigt.

Programmausdruck D.5: **Klassendeklaration zum bequemen Setzen und Entgegennehmen von *FSDatasets* in *FSMesh*-Objekten.**

```
1   class UnstructDatasetHandler:
2       """
        used to ease the getting and setting of unstructured dataset in
4       a FSMesh object. It returns a dataset based on a given name as
        Numpy array or sets the values in a dataset of given name equal
6       to the values in the given Numpy array
        """
8       def __init__(self, mesh):

10          import FSNumPyInterface
            from FSNumPyArray import FSNumPyArray
12
            self.mesh = mesh
14          self.clac = self.mesh.GetClac()
            self.arrayConverter = FSNumPyArray(self.clac)
16
        def Get(self, datasetName):
18          dataset = self.mesh.GetUnstructDataset(datasetName)
            values = dataset.GetValues()
20          valuesNumpy = self.arrayConverter.Convert2Numpy(values)
```

```
        return valuesNumpy

    def Set(self, datasetName, datasetValuesNumpy):
        dataset = self.mesh.GetUnstructDataset(datasetName)
        datasetValues = dataset.GetValues()
        newValues = self.arrayConverter.Convert2FSFloatArray(datasetValuesNumpy)
        datasetValues.CopyData(newValues)
```

D.4 Controller für *externe* Bewegungsvorgabe in TAU-Simulation

Programmausdruck D.6: **Hilfsklasse zur TAU-externen Verwaltung der Bewegungszustände mehrerer Zeitebenen und zur Übergabe der Bewegungszustände an TAU**

```
import FSDM
from FSDataManager import FSLog

class TauMotionController:
    '''
    '''
    def __init__(self, clac, TauExtMotionInterface):
        self.clac = clac
        self.tauExtMotion = TauExtMotionInterface
        self.motionStringStack = {"oold":None, "old":None, "actual":None}

        class MotionChecker:
            ''' Convient class for checking whether the motion data
            of two consecutive time steps are identical
            '''
            ALMOST_ZERO = 1.0e-12

            def __init__(self):
                self.motionData = None
                self.motionDataOld = None
                self.initiallySet = False

            def UpdateMotion(self, motionData):
                if not self.initiallySet:
                    self.motionDataOld = motionData
                    self.initiallySet = True
                else:
                    self.motionDataOld = self.motionData
                self.motionData = motionData

            def HasMotionChanged(self):
                if self.initiallySet:
                    diff = [abs(self.motionData[i]-self.motionDataOld[i]) \
                                < self.ALMOST_ZERO \
```

```
35                                  for i in range(len(self.motionData))]
                    if False in diff:
37                      return True
                else:
39                  return False

41          self.motionChecker = MotionChecker()

43      def GetClac(self):
            return self.clac
45
        def UpdateMotionDataInLocalStack(self, nodeName, motionString):
47          FSLog(self.GetClac(), 0, "Updating motion data on local stack")
            motionStringList = motionString.split()
49          motionData = map(float, motionStringList)
            self.motionChecker.UpdateMotion(motionData)
51          # actual set-operation in local motion stack
            self.motionStringStack["actual"] = "%s %s" %(nodeName, motionString)
53
        def GetMotionStringFromLocalStackForTimeLevel(self, timeLevel="actual"):
55          if timeLevel in self.motionStringStack:
                return self.motionStringStack[time_level]
57          else:
                return None
59
        def UpdateMotionInTauFromLocalStack(self):
61         FSLog(self.GetClac(), 0, "Updating motion data in TAU from local stack")
           for timeLevel in "oold", "old", "actual":
63              motionString = \
                      self.GetMotionStringFromLocalStackForTimeLevel(timeLevel)
65              if not motionString is None:
                      self.tauExtMotion.update_motion(motionString,
67                                                    time_level=timeLevel,
                                                      verbose=True,
69                                                    update_chimera=False)

71      def PushMotionDataOnLocalStack(self):
            FSLog(self.GetClac(), 0, "Pushing motion data on local stack")
73          self.motionStringStack["oold"] = self.motionStringStack["old"]
            self.motionStringStack["old"] = self.motionStringStack["actual"]
75
        def GetAeroLoadsForNode(self, nodeName):
77          return self.tauExtMotion.get_forces_and_moments_for_node(nodeName)

79      def GetAeroLoadsFromMonitor(self):
            return self.tauExtMotion.get_forces_and_moments()
81
        def HasMotionDataChanged(self):
83          return self.motionChecker.HasMotionChanged()
```

E Koordinatensysteme

In diesem Abschnitt werden die auf der Seite von Simulink und TAU verwendeten Koordinatensystem beschrieben. Die in den Tabellen B.1 und B.2 aufgelisteten Daten, die zwischen Simulink und TAU ausgetauscht werden, beziehen sich auf die nachfolgend gezeigten Koordinatensysteme.

Die Koordinatensysteme der Abbildungen E.1 und E.2 werden innerhalb des Simulink-Modells verwendet, die in den Abbildungen E.3 bis E.5 gezeigten Koordinatensystem kommen auf TAU-Seite zum Einsatz.

Vor der Übertragung der aerodynamischen Kräfte und Momente an Simulink werden diese jeweils auf der Seite des CFD-Prozesses in das Koordinatensystem aus Abb. E.2 transformiert. Dazu wird die von TAU berechnete Kraft-/Momentengruppe zunächst in die Flugzeugnase (Ursprung des Koordinatensystems aus Abb. E.2) verschoben und danach die Transformation bzgl. etwaiger unterschiedlicher Orientierungen der Simulink- und TAU-seitig verwendeten Systeme vorgenommen[1]. Zur Transformation wird die Transformationsmatrix $M_{TAU,b-SLK,b}$ aus (E.1) mit den Euler-Winkeln $\phi = 0$, $\theta = \pi$, und $\psi = 0$ verwendet:

$$M_{TAU,b-SLK,b} = \begin{pmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi \in \theta\sin\psi + \cos\phi\cos\psi & \sin\phi\cos\theta \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\cos\theta \end{pmatrix}. \quad \text{(E.1)}$$

[1] Dieser Punkt tritt nur in Kraft, falls bei TAU die Ausgabe der Kraft-/Momentengruppe im System aus Abb. E.4 erfolgt, d. h. der TAU-Parameter **`Reference system of forces and moments (tau/ln9300)`** ist auf **`tau`** statt auf **`ln9300`** eingestellt ist.

Abbildung E.1: **Geostationäres Koordinatensystem, das im Simulink-Modell verwendet wird (Subskript *SLK,g*). Der Ursprung liegt in der Nasenspitze des FMTA. Die z-Achse steht senkrecht auf der Horizontalebene und zeigt in Richtung des Erdmittelpunkts. Die x- und y-Achsen liegen innerhalb der Horizontalebene. Dabei zeigt die y-Achse in Richtung des rechten Flügels. Die Orientierung des Systems entspricht der LN9300-Definition.**

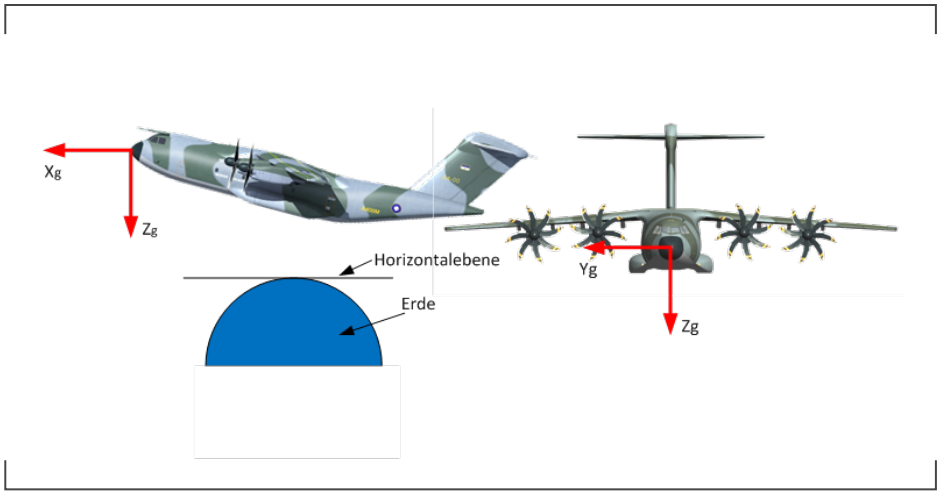


Abbildung E.2: **Körperfestes Koordinatensystem, das im Simulink-Modell verwendet wird (Subskript *SLK,b*). Es greift ebenfalls in der Flugzeugnase des FMTA an. Die Flugzeuglängsachse wird als x-Achse definiert und in Flugrichtung positiv gezählt. Die y-Achse zeigt in Flugrichtung nach rechts. Die z-Achse steht senkrecht auf der von x- und y-Achse aufgespannten Ebene. Die Orientierung des Systems entspricht der LN9300-Definition.**

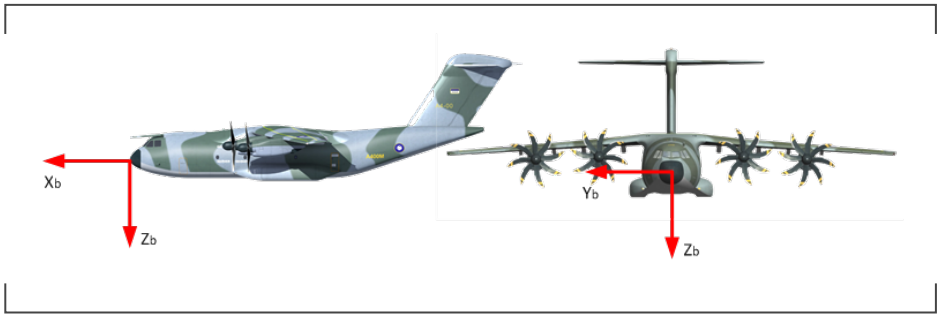


Abbildung E.3: **TAU-Netzkoordinatensystem (Subskript *TAU,0*). Es entspricht dem Koordinatensystem, in dem das CFD-Netz vorliegt. Im Fall der Netze für die FMTA-Konfiguration hat das System seinen Ursprung an der Flugzeugnase, die x-Achse verläuft entlang der Flugzeuglängsachse und wird entgegen der Flugrichtung negativ gezählt. Die y-Achse zeigt in Flurichtung nach rechts. Die z-Achse steht senkrecht auf x- und y-Achse und zeigt nach oben.**

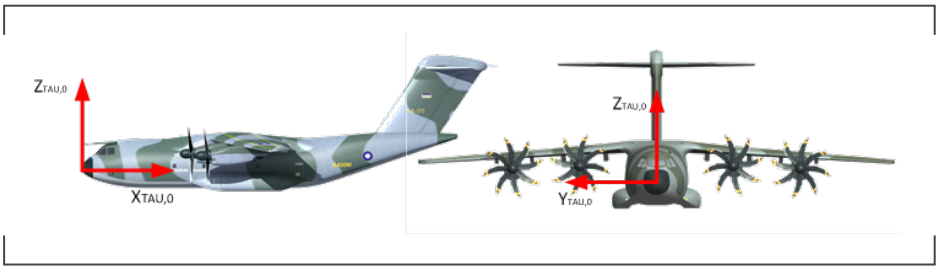


Abbildung E.4: **Körperfestes Koordinatensystem, das von TAU intern verwendet wird (Subskript *TAU,b*). Es ist relativ zum Flugzeug genauso orientiert wie das System aus Abb. E.3. Der Ursprung des Systems ist allerdings um den Vektor $\vec{b} = (b_{TAU,x}, b_{TAU,y}, b_{TAU,z})^T$ gegenüber dem System aus Abb. E.3 verschoben. Das Flugzeug wird als fest verbunden mit dem Punkt *b* aufgefasst.**

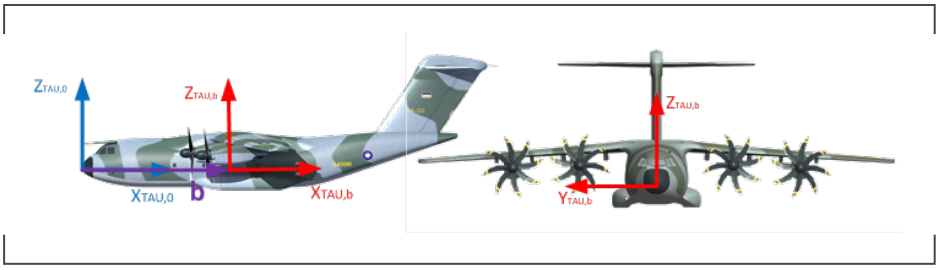
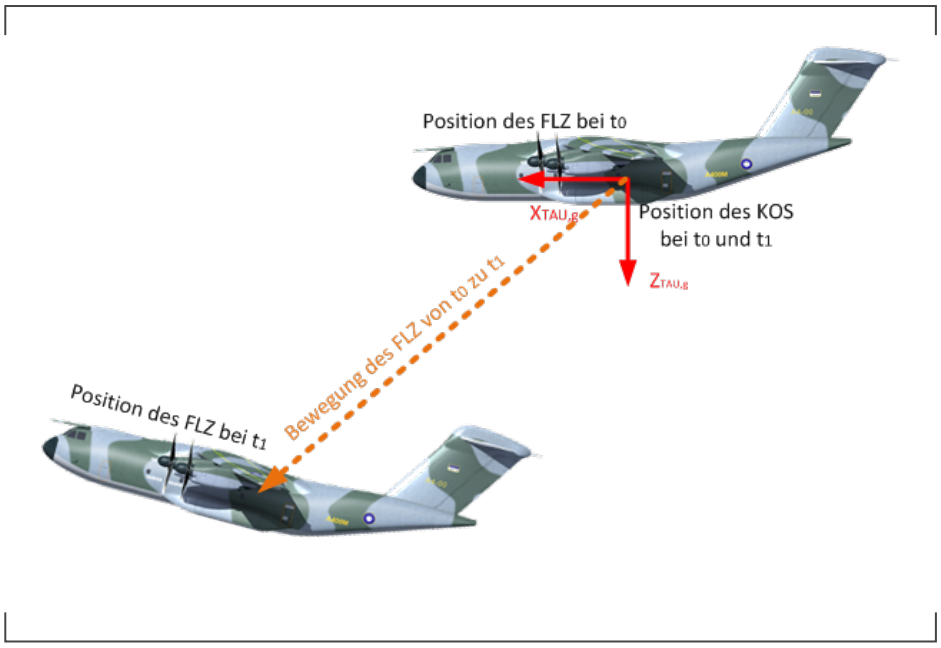


Abbildung E.5: **Geostationäres und körperfestes TAU-Koordinatensystem mit LN9300-Ausrichtung (Subskript *TAU,g*). Zum Zeitpunkt $t = 0$ fallen die Ursprünge beider Systeme mit dem Punkt *b* des Flugzeugs zusammen (siehe Abb. E.4). Für $t \neq 0$ verbleit das geostationäre System an seiner Ausgangsposition, während das körperfeste System sich fest im Punkt *verankert* mit dem Objekt mitbewegt, wobei es allerdings seine Anfangsorientierung beibehält.**



F Software-Voraussetzungen

Die Nutzung der im vorliegenden Bericht beschriebenen CFD-Simulink-Kopplung beruht darauf, dass die folgende Software verfügbar ist:

- TAU (Release 2014.1.0)
- FSTau/FSTauInterface
- FSDM
- FSDeformation/FSWallDistance
- FSNumPyInterface
- Python-Interpreter $\geq$ 2.7.5
- NumPy $\geq$ 1.3.0
- MATLAB (Version 2007) mit Simulink-Erweiterung

Literaturverzeichnis

[1] Jann, T. et al. (2014): "MiTraPor II – Abschlussbericht", DLR IB-111-2014/23, DLR Institut für Flugsystemtechnik, Braunschweig (Ref. auf Seiten 5, 7).

[2] Schwamborn, D., Gerhold, T., Heinrich, R. (2006): "The DLR TAU-Code: Recent Applications in Research and Industry", *European Conference on Computational Fluid Dynamics ECCOMAS CFD*, Sept. (Ref. auf Seite 6).

[3] Langer, S., Schwöppe, A., Kroll, N. (2014): "The DLR Flow Solver TAU – Status and Recent Algorithmic Developments", AIAA-2014-0080, *52nd AIAA Aerospace Sciences Meeting*, 13.-17. Jan., National Harbor, MD, USA (Ref. auf Seite 6).

[4] Stickan, B. (2009): "Implementation and Extension of a Mesh Deformation Module for the Parallel FlowSimulator Software Environment", Diplomarbeit, RWTH Aachen, Lehrstuhl für Computergestützte Analyse Technischer Systems und Airbus Deutschland GmbH, Bremen, Abteilung *Tools and Simulation* (Ref. auf Seite 19).

[5] Barnewitz, H., Stickan, B. (2013): "Improved Mesh Deformation", In: Eisfeld, B., Barnewitz, H., Fritz, W., Thiele, F. (eds.): "Management and Minimisation of Uncertainties and Errors in Numerical Aerodynamics", *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, Springer Berlin Heidelberg, Vol. 122, pp. 219-243

[6] Stickan, B., Barnewitz, H., Hansen, L.-U. (2013): "User Guide FSDeformation Module", (Verfügbar im SVN-Repositorium von *FSDeformation* unter `https://dev2.as.dlr.de/svn/fsdeformation/trunk/FSDeformation/UserGuide/UserGuide.pdf`) (Ref. auf Seiten 19, 42).

[7] Reimer, L. (2013): "User guide of FSCtrlSurfDisplFieldGenerator", (Verfügbar im SVN-Repositorium von *FSDeformation* unter `https://dev2.as.dlr.de/svn/fsdeformation/trunk/FSCtrlSurfDisplFieldGenerator/doc/User guide of FSCtrlSurfDisplFieldGenerator.pdf`) (Ref. auf Seite 44).

[8] Einarsson, G. et al. (2014): "TAU-Python Interface Guide for Release 2014.1.0", (Ver-

fügbar im SVN-Repositorium von *TAU* unter `https://tausvn.as.dlr.de/repos/dlr/tau/trunk/taudir/doc/taupython_doc.pdf`) (Ref. auf Seite 22).

IB 124-2014/910

Eine TAU-Simulink-Kopplung zur aerodynamisch-flugdynamischen Simulation – Implementierung und Anwendung auf eine generische militärische Transportflugzeugkonfiguration mit Beschädigung

Lars Reimer, Christian Gall, Sven Geisbauer

Verteiler:

Institut für Aerodynamik und Strömungstechnik (BS)	1	Exemplar
Institut für Flugsystemtechnik (BS)	1	Exemplar
Verfasser/Co-Autoren	3	Exemplare
Institutsleitung AS	1	Exemplar
Abteilungsleiter $C^2A^2S^2E$	1	Exemplar
Gruppenleiter	1	Exemplar
DLR Zentralbibliothek Braunschweig	2	Exemplare
Reserve	1	Exemplar
	11	Exemplare